FACULDADE DE ENGENHARIA DA UNIVERSIDADE DO PORTO

Criação de *framework* REST/HATEOAS *Open Source* para desenvolvimento de APIs em Node.js

Filipe Perdigão de Sousa

DISSERTAÇÃO

Mestrado Integrado em Engenharia Informática e Computação

Orientador: Ademar Manuel Teixeira de Aguiar

20 de Julho de 2015

Criação de *framework* REST/HATEOAS *Open Source* para desenvolvimento de APIs em Node.js

Filipe Perdigão de Sousa

Mestrado Integrado em Engenharia Informática e Computação

Aprovado em provas públicas pelo Júri:

Presidente: Jorge Barbosa

Arguente: Ricardo Machado

Orientador: Ademar Aguiar

20 de Julho de 2015

Resumo

O aparecimento do estilo de arquitetura *Representational State Transfer* (REST) tornou possível o surgimento de mais uma alternativa para a implementação de serviços *Web*, tentando que, de um modo mais simples, fosse possível fazer a construção de uma *Application Programming Interface* (API) para a comunicação entre os diferentes componentes de um sistema, nomeadamente, entre o servidor e os diferentes clientes.

Com isto, surgiram então inúmeras implementações, compatíveis com as mais variadas linguagens de programação, tal como Java, Python, Ruby, Scala, JavaScript, entre outras, de modo a proporcionar aos programadores uma forma acessível dos mesmos implementarem os seus serviços com base nesta arquitetura. Atualmente a tecnologia Node.js é fortemente utilizada para o desenvolvimento de aplicações *Web*, nomeadamente para a implementação deste tipo de serviços. É neste sentido que surgiu este tema de dissertação, o desenvolvimento de uma *framework* para Node.js, que permita oferecer aos criadores de APIs REST uma forma de criarem os seus serviços seguindo as boas práticas e as tendências tecnológicas da linguagem JavaScript e da comunidade envolvida, permitindo assim oferecer uma solução tecnologicamente competitiva.

Como um dos principais objetivos e pontos de inovação, a *framework* desenvolvida pretende dar a capacidade de os programadores poderem seguir as boas práticas e restrições aconselhadas para serviços deste tipo de uma forma praticamente transparente, permitindo que de um modo simples e de certo modo automático, determinados conceitos lhe sejam incutidos propiciando o desenvolvimento de serviços da forma mais correta possível. Apesar dos aspetos que são automaticamente incutidos nos serviços, a *framework* também pretende oferecer ao programador alguma liberdade tecnológica, permitindo-lhes tomar decisões ao nível de alguns componentes necessários ao serviço.

Outro grande foco é a capacidade de desenvolvimento de serviços REST que sejam capazes de integrar com aplicações desenvolvidas através da abordagem *NoBackend*, nas quais as tarefas relativas ao *backend* são abstraídas no desenvolvimento dos respetivos clientes. Através desta funcionalidade é esperada ainda a possibilidade do próprio serviço gerar automaticamente código de cliente que possa ser integrado nas aplicações em questão.

Além disso, esta dissertação tem ainda como objetivo o desenvolvimento de uma *framework* que tenha ao seu alcance mecanismos de serialização de dados em formato JSON, documentação automática de recursos e ainda a geração automática de formulários *web* para testes da API.

Por último, todo o trabalho desenvolvido será ainda disponibilizado com a comunidade *open-source* por forma a validar não só os conceitos introduzidos mas também a sua implementação.

Abstract

The appearing of the Representational State Transfer (REST), an architectural pattern, came to improve the development context of web services with an alternative for the implementation of computer distributed systems, attempting that, through a simple way, it was possible to develop an Application Programming Interface (API) for communication between the system's components, particularly between the server and the different available clients.

Allied to this fact, there were so many implementations, compatible with various programming languages, like Java, Python, Ruby, Scala, JavaScript, and others, in order to provide developers an easy way to implement their services, attending this architecture. Nowadays Node.js technology is highly used for web applications development, in particular for the implementation of this type of services. The theme of this dissertation appears in this context, the development of a Node.js framework which allows to offer creators of this kind of service a way to create REST APIs following the best practices and technology trends of language and community involved, thus offering a technologically competitive solution.

As one of the main objectives and innovation points, the framework intends to give developers the ability to follow the standards and recommended restrictions for such services from an almost transparent way, allowing that easily and automatically some concepts are instilled providing the services development as correctly as possible. Despite the features that are automatically instilled in services, the framework also aims to offer to programmers some technological freedom, allowing them to make their own decisions about some components required for the service.

Another main objective is the capability of services implementation that allows the integration with applications developed in a NoBackend approach, in which the backend tasks are abstracted in the development of respective clients. This feature also allows the automatically generation of client code that can be included in their clients.

Moreover, this dissertation also has as the objective, of developing of a framework that offers data serialization mechanisms in JSON format, automatic API documentation, logging, analytic and also the automatic generation of web forms for API test.

In the end, all produced work will be published to open-source community in a way to validate both introduced concepts as their implementation.

Agradecimentos

Em primeiro lugar eu gostaria de agradecer a toda a equipa da Glazed Solutions, empresa que proporcionou o desenvolvimento desta dissertação, por me acolherem como um membro durante esta etapa. Ao Engenheiro Pedro Campos um especial agradecimento pelo acompanhamento, dedicação e por toda a passagem de conhecimento, sem o qual a conclusão deste trabalho não seria possível.

Ao meu orientador, professor Ademar Aguiar quero agradecer pelo acompanhamento e pela disponibilidade manifestada.

Gostaria também de manifestar a minha profunda gratidão tanto aos meus pais como à minha irmã por todo o suporte manifestado ao longo destes anos e por terem permitido que um dos meus objetivos de vida fosse alcançado.

À minha namorada, Sandra Borges, um especial agradecimento pelo apoio constante bem como pela paciência e atenção durante esta etapa da minha vida.

Quero agradecer também a todos os meus amigos que me acompanharam durante estes anos, e em especial ao Cristiano Alves e ao José Magalhães pelo seu apoio mas também por todos os momentos de descontração.

Por último mas não menos importante, agradeço a toda a minha família, que ao longo destes anos sempre esteve ao meu lado e sempre me fizeram acreditar no alcance desta etapa da minha vida.

Filipe Perdigão de Sousa

Conteúdo

1 Introdução **1**
- 1.1 Contexto . . . 1
- 1.2 Motivação e Objetivos . . . 2
- 1.3 Estrutura da Dissertação . . . 3

2 Estado da Arte **5**
- 2.1 Serviços *Web* . . . 5
  - 2.1.1 Contexto Histórico . . . 5
  - 2.1.2 SOAP . . . 7
  - 2.1.3 REST . . . 8
- 2.2 JavaScript . . . 12
  - 2.2.1 Ecmascript 6 . . . 13
  - 2.2.2 EcmaScript 7 . . . 15
  - 2.2.3 Node.js . . . 16
- 2.3 REST *Frameworks* . . . 17
  - 2.3.1 Django REST Framework . . . 17
  - 2.3.2 Flask-RESTful . . . 18
  - 2.3.3 Restlet . . . 18
  - 2.3.4 Spark . . . 19
  - 2.3.5 Sinatra . . . 19
  - 2.3.6 Express . . . 20
  - 2.3.7 Restify . . . 20
  - 2.3.8 SailsJS . . . 21
  - 2.3.9 LoopBack . . . 21
  - 2.3.10 Comparação de *frameworks* . . . 22
- 2.4 Documentação Automática . . . 24
  - 2.4.1 Swagger . . . 24
  - 2.4.2 Slate . . . 24
  - 2.4.3 API Blueprint . . . 25
  - 2.4.4 RAML . . . 25
  - 2.4.5 I/O Docs . . . 25
- 2.5 *NoBackend* . . . 26
  - 2.5.1 Parse . . . 26
  - 2.5.2 Hoodie . . . 27
  - 2.5.3 Meteor . . . 28
  - 2.5.4 BaasBox . . . 28
  - 2.5.5 Comparação . . . 29
- 2.6 Micro Serviços . . . 30

2.6.1 Motivação . . . 31
2.6.2 Vantagens . . . 31
2.6.3 Desvantagens . . . 32
2.7 Resumo . . . 33

3 Problema e Solução **35**
3.1 Descrição do Problema . . . 35
3.1.1 Arquitetura REST . . . 36
3.1.2 Interface Uniforme . . . 36
3.1.3 Documentação do serviço . . . 36
3.1.4 *HATEOAS* . . . 36
3.1.5 Performance . . . 37
3.1.6 *NoBackend* . . . 37
3.1.7 Comunicação em tempo real . . . 37
3.1.8 Tecnologias . . . 37
3.2 Solução . . . 38
3.2.1 Narrativas de Utilização . . . 39
3.2.2 *Dreamcode* . . . 40
3.2.3 Arquitetura . . . 44

4 Implementação **53**
4.1 Tecnologias Utilizadas . . . 53
4.1.1 JavaScript . . . 53
4.1.2 Transpiladores de Javascript . . . 54
4.1.3 Base de dados . . . 55
4.2 Metodologia . . . 55
4.3 Funcionalidades Implementadas . . . 56
4.4 Detalhes de implementação . . . 57
4.4.1 *Resources* . . . 58
4.4.2 *Models* . . . 64
4.4.3 Router . . . 65
4.4.4 Validação de dados . . . 67
4.4.5 Relações . . . 69
4.4.6 Documentação . . . 71

5 Validação **75**
5.1 Validação de Funcionalidades . . . 75
5.1.1 Testes Unitários . . . 75
5.1.2 Prova de Conceito . . . 76
5.2 Comparação com soluções existentes . . . 77
5.2.1 Mapeamento pedidos HTTP . . . 77
5.2.2 Acesso à base de dados . . . 80
5.2.3 Autenticação . . . 81
5.2.4 Autorização . . . 82
5.3 Feedback da comunidade . . . 84

6 Conclusões **85**
6.1 Avaliação das escolhas tecnológicas . . . 85
6.2 Avaliação do trabalho desenvolvido . . . 86
6.3 Trabalho Futuro . . . 88

Referências **89**

A Comparação SOAP vs REST **93**

B Métodos e códigos de estado HTTP **95**
B.1 Métodos . . . 95
B.2 Códigos de Estado . . . 96
B.2.1 Informativo . . . 96
B.2.2 Sucesso . . . 96
B.2.3 Redirecionamento . . . 96
B.2.4 Erro de cliente . . . 97
B.2.5 Erro de Servidor . . . 97

C Comparação de *frameworks* REST **99**

D Relatório de Cobertura de Código **101**

E Inicialização de serviços REST **103**
E.1 Koa . . . 103
E.2 Hapi . . . 104
E.3 Solução implementada . . . 105

CONTEÚDO

Lista de Figuras

2.1 Arquitetura de um serviço SOAP 8
2.2 Processamento das operações em Node.js 16
2.3 Arquitetura de um serviço *NoBackend* 27
2.4 Possível arquitetura de um sistema baseado em micro serviços 30

3.1 Visão geral da arquitetura da *framework* 44
3.2 Relação entre utilizadores e grupos de permissões 48

4.1 Metodologia de implementação da solução 56
4.2 Organização das entidades implementadas 57
4.3 Documentação interativa através de Swagger 73

5.1 Esquema de dados da API implementada como prova de conceito 76

D.1 Relatório de cobertura de código 101

Lista de Tabelas

2.1 Tabela comparativa de *frameworks* REST resumida 22
2.2 Comparação de *frameworks* de *NoBackend* 29

C.1 Tabela comparativa de *frameworks* REST - Parte 1 99
C.2 Tabela comparativa de *frameworks* REST - Parte 2 100

Abreviaturas e Símbolos

API	Application Programming Interface
COM	Component Object Model
CORBA	Common Object Request Broker Architecture
CORS	Cross-Origin Resource Sharing
CRUD	Create/Read/Update/Delete
DCOM	Distributed Component Object Model
HATEOAS	Hypermedia as the Engine of Application State
HTML	Hypertext Markup Language
HTTP	Hypertext Transfer Protocol
IDE	Integrated Development Environment
JAX-RS	Java API for RESTful Web Services
JSON	JavaScript Object Notation
JSP	JavaServer Pages
JVM	Java Virtual Machine
ORB	Object Request Broker
ORM	Object-relational Mapping
PLIST	Property List
RAML	RESTful API Modeling Language
REST	Representational State Transfer
RPC	Remote Procedure Call
SDK	Software Development Kit
SMTP	Simple Mail Transfer Protocol
SOAP	Simple Object Access Protocol
TCP	Transmission Control Protocol
TDD	Test Driven Development
UDDI	Universal Description Discovery and Integration
URI	Uniform Resource Identifier
URL	Uniform Resource Locator
WSDL	Web Service Definition Language
XML	eXtensible Markup Language
YAML	Yaml Ain't Markup Language

Capítulo 1

Introdução

Nos últimos anos, com a evolução da tecnologia e o seu acesso cada vez mais facilitado, a sociedade foi habituada a utilizar as aplicações informáticas como forma de gestão e manipulação de dados relativos às mais variadas áreas de trabalho, passando assim a ser cada vez mais importante a evolução da forma como estas capacidades estariam disponíveis aos utilizadores.

Com a evolução da Internet e o respetivo aumento da sua utilização, os utilizadores, até então habituados a utilizar aplicações instaladas localmente, nas suas próprias máquinas, cujo acesso a dados se restringia aos limites do seu ambiente de execução, foram cada vez mais confrontados com alterações nos paradigmas deste tipo de aplicações, passando as mesmas a estarem disponíveis de uma forma descentralizada. Com isto, cada vez mais as aplicações locais, cuja componente de comunicação com outros componentes distribuídos era praticamente nula, foram sendo substituídas por outras que nos oferecem os seus serviços de uma forma remota aos quais é possível aceder independentemente da localização, quer através de um computador como através de outros tipos de dispositivos, como *tablets* ou *smartphones*.

Denominados de serviços *Web*, estes tipos de serviços podem ser implementados com base em dois tipos de arquiteturas distintas, nomeadamente através de arquiteturas baseadas em *Simple Object Access Protocol* (SOAP), ou através de arquiteturas baseadas em *Representational State Transfer* (REST), centrando-se esta dissertação nesta última arquitetura.

1.1 Contexto

Esta dissertação insere-se no contexto de desenvolvimento de serviços *web* com base na arquitetura REST. Esta arquitetura, apresentada por Roy Thomas Fielding, em 2000, na sua tese de doutoramento [Fie00], surgiu como uma forma alternativa à implementação de serviços *web*, permitindo a disponibilização dos diferentes tipos de recursos de um serviço de uma forma simplificada, através da utilização de protocolos já utilizados no âmbito da Internet e *standards* flexíveis, dando alguma liberdade aos programadores na implementação dos seus serviços.

Neste momento, é uma das arquiteturas de eleição para o desenvolvimento de serviços distribuídos [Ros13], tentando possibilitar a implementação simplificada de APIs para a

comunicação entre os diferentes componentes do sistema, nomeadamente entre o servidor e os diferentes clientes.

Por forma a auxiliar os programadores no desenvolvimento deste tipo de serviços, surgiram *frameworks*[1] para as mais variadas linguagens, tais como Java, Python, Ruby, .Net e JavaScript, que permitem encapsular as especificidades referentes a esta arquitetura e oferecer aos programadores uma forma simplificada e, por vezes, transparente de implementarem este tipo de serviços. Embora focada numa perspetiva de desenvolvimento através da linguagem JavaScript, nomeadamente através da plataforma de desenvolvimento de aplicações *web* Node.js, esta dissertação insere-se neste contexto, no desenvolvimento de uma *framework* REST.

Tendo por base a arquitetura REST, esta dissertação incide no estudo de algumas das soluções já existentes nas diversas linguagens de programação, focando-se na utilização dos *standards* definidos pela arquitetura, mas também no estudo das boas práticas e tendências relacionadas não só com a componente dos serviços distribuídos mas também com as tendências e tecnologias usadas pela comunidade envolvida no desenvolvimento deste tipo de serviços.

Esta dissertação foi desenvolvida em ambiente empresarial, na empresa Glazed Solutions. A Glazed Solutions é uma empresa de *software* para *smartphones* fundada no início de 2011, especializada no desenvolvimento de aplicações móveis para plataformas como iPhone/iPad (iOS), Android, Windows Phone 7, Bada, WebOS. Nesta empresa, a investigação e a tecnologia de ponta têm papéis proeminentes, sendo neste sentido que se encaixa o desenvolvimento desta dissertação.

1.2 Motivação e Objetivos

Atualmente a tecnologia Node.js está a ser cada vez mais utilizada para o desenvolvimento de aplicações *web*, nomeadamente para a implementação da componente de servidor dos sistemas distribuídos, que será acessível aos diferentes tipos de clientes do sistema. Apesar de esta tecnologia ser ainda recente, existem já algumas *frameworks* que permitem aos programadores implementarem as suas APIs REST de uma forma simplificada e tendo em conta os *standards* definidos pela arquitetura. No entanto, a oferta, relativamente a soluções *open-source* e que ao mesmo tempo permitam ter em conta tanto as boas práticas desta arquitetura, como as tendências desta tecnologia, ainda é reduzida.

É neste sentido que surgiu este tema de dissertação, o desenvolvimento de uma solução em Node.js que permita a implementação de serviços REST de uma forma simplificada e facilmente compreensível através da utilização de um mínimo de conceitos possível. Ao mesmo tempo é pretendido oferecer aos programadores flexibilidade na escolha de algumas componentes tecnológicas mas também uma interface amigável através da qual é possível seguir tanto os *standards* e boas práticas enunciadas para este tipo de arquiteturas mas também para a utilização das tendências atuais do desenvolvimento através de JavaScript.

Um dos problemas muitas vezes encontrados em *frameworks* deste tipo é a forma como estas se apresentam aos programadores e a forma como estes são obrigados a interagir as mesmas.

[1] Conjunto de componentes de *software* que se unem com foco na execução de uma operação maior

Com foco na redução do tempo necessário para a compreensão da ferramenta, é pretendido que a *framework* seja organizada de um modo coerente nos vários níveis do serviço, oferecendo uma interface simples, objetiva e ao mesmo tempo unificada desde o acesso aos dados até ao seu tratamento nos controladores do serviço.

Além dos objetivos já mencionados, esta dissertação inclui também nos seus principais objetivos a implementação de uma *framework* que seja capaz de ser integrada em aplicações cujo desenvolvimento foge à abordagem tradicional. É pretendido que esta dissertação possa ser integrada na implementação de sistemas onde a componente de cliente possui um maior relevo, através da qual são abstraídas as diferentes tarefas de servidor. Para isto, é pretendido que além da *framework* ter a capacidade de disponibilizar uma interface genérica para a gestão dos diferentes recursos seja também capaz de oferecer a possibilidade de utilização de código de cliente como forma de abstração da utilização do serviço.

Pretende-se ainda que a solução desenvolvida seja capaz de oferecer funcionalidades como a documentação automática de recursos, geração automática de formulários de teste das APIs, mecanismos *cache* focados na performance dos serviços e mecanismos de transmissão de dados em tempo real.

Como forma de concretização dos objetivos enumerados anteriormente será feito o estudo aprofundado das bases teóricas relativas a arquiteturas REST, aos seus princípios, restrições bem como à sua aplicabilidade no desenvolvimento de serviços atuais. Além disso, será feita uma análise de algumas soluções já existentes nas várias linguagens, bem como o estudo do estado da arte relativamente a tecnologias *NoBackend* e às soluções de documentação de APIs já existentes. Em paralelo será também estudada a linguagem JavaScript, as boas práticas, tendências e conceitos mais recentes bem como a tecnologia Node.js.

Assim como forma de desenho de uma solução completa nas várias áreas de estudo, serão analisadas as várias componentes e funcionalidades de interesse para o desenvolvimento de serviços deste tipo bem como o estudo da melhor forma de as mesmas interagirem entre si com foco na implementação de uma solução coesa e ao mesmo tempo preparada para os diferentes tipos de abordagens.

1.3 Estrutura da Dissertação

Para além deste capítulo, referente à introdução, este relatório contém mais 5 capítulos.

No capítulo 2 são descritas as diferentes arquiteturas que possibilitam a construção de serviços de uma forma distribuída, bem como apresentada a análise das diferentes soluções já existentes para a implementação de APIs REST, bem como para a implementação de serviços através de uma abordagem de *NoBackend*. Além disso são apresentadas também as técnicas de documentação deste tipo de serviços já existentes bem como os conceitos relacionados com arquiteturas de micro serviços. Neste capítulo é ainda apresentada a linguagem de programação a ser utilizada bem como alguns aspetos caraterísticos da mesma.

No capítulo 3 é apresentada uma descrição mais pormenorizada dos problemas encontrados no estudo do estado da arte das diferentes temáticas na qual se insere o desenvolvimento desta dissertação. Além disso é descrita a proposta de solução para a resolução dos problemas encontrados e também apresentado o processo através do qual foi possível ir ao encontro das necessidades da solução, desde a especificação das narrativas de utilização à definição do código ideal da solução.

No que diz respeito ao capítulo 4, este é o capítulo onde são enumeradas as diferentes funcionalidades e módulos desenvolvidos bem como os vários detalhes relativos à sua implementação.

No capítulo 5 é apresentada a forma como a solução implementada foi validada, de modo a provar o sucesso ou insucesso desta dissertação.

Por último, no capítulo 6 é apresentado um resumo do trabalho desenvolvido bem como as principais conclusões retiradas do desenvolvimento desta dissertação. Além disso são também apresentadas as linhas condutoras para o trabalho futuro de modo a que o trabalho desenvolvido nesta dissertação possa evoluir atendendo tanto às funcionalidades que não foram desenvolvidas como a outros aspetos que seriam interessantes ser tomados em consideração numa possível evolução da arquitetura.

Capítulo 2

Estado da Arte

No decorrer deste capítulo irão ser abordados com maior profundidade os conceitos e tecnologias referidas no capítulo anterior. Deste modo, este capítulo serve como uma representação da base de conhecimento necessária ao desenvolvimento desta dissertação, para que possa ser feito um estudo não só dos conceitos necessários à correta compreensão dos objetivos mas também da base tecnológica já existente na área em estudo. Tem-se então como um dos principais objetivos a análise do estado da arte por forma a salientar os problemas existentes nas soluções atuais mas também ter a perceção dos pontos positivos existentes por forma a ter uma base de apoio de boas práticas e aspetos a considerar na concretização desta dissertação.

2.1 Serviços *Web*

2.1.1 Contexto Histórico

No final dos anos 90, com a crescente utilização da Internet, e com a explosão dos serviços *Web*, muitas empresas sentiram a necessidade de expor os seus serviços para que estes tivessem disponíveis para outras entidades. Com isto, surgiu a necessidade de criação de uma forma de comunicação entre diferentes aplicações, de modo a permitir que aplicações alojadas em diferentes máquinas ou até em diferentes redes fossem capazes de comunicar entre si. Por forma a atender a esta situação foram propostas diferentes tecnologias alternativas, tais com *Remote Procedure Call* (RPC), *Common Object Request Broker Arquitecture* (CORBA) e *Distributed Component Object Model* (DCOM).

RPC

Relativamente ao conceito de RPC, este refere-se a um paradigma que atribui aos programas a capacidade de interação com outros através da rede, podendo um determinado programa invocar métodos num outro espaço de endereçamento, como é o caso de execução em outros computadores. Com base na ideia de que as chamadas a procedimentos dentro de um único

computador são bem conhecidas e bem percebidas, é proposto um mecanismo que atribui essas mesmas características à comunicação através da rede [BN84].

Através deste mecanismo, quando um método é invocado por um determinado processo, as tarefas vão ser executadas na máquina de destino, enquanto o processo inicial é suspenso à espera da respetiva resposta. Posteriormente, quando esta estiver disponível, é devolvida ao processo inicial e este continua a sua execução normalmente.

Deste modo, é possível fazer a implementação de um sistema distribuído de forma quase transparente para o programa cliente e semelhante ao que acontece na implementação de um programa local.

CORBA

A arquitetura CORBA, criada pelo *Object Management Group*, tem como objetivo especificar um método de permitir a comunicação entre diferentes tipos de aplicações, em diferentes ambientes e linguagens de desenvolvimento, através da Internet.

Esta arquitetura segue os conceitos de *Object Request Broker* (ORB), existindo um objeto intermediário, que oferece aos clientes do sistema a possibilidade de comunicarem com o programa ou objeto servidor de um modo transparente para os utilizadores, sem a necessidade de conhecer a localização ou as especificações do mesmo [Rou06].

O ORB é responsável pela interação entre as diferentes componentes distribuídas do sistema, recebe os pedidos dos programas clientes, reencaminha para os objetos remotos competentes e assim que a resposta estiver disponível é também o responsável por enviá-la para o cliente.

DCOM

No que diz respeito a DCOM, tecnologia desenvolvida pela Microsoft, possui foco na comunicação entre objetos alojados em diferentes computadores, quer localizados na mesma rede local como em outros pontos acessíveis através da Internet [HK97]. A ideia base desta tecnologia é a utilização de um objeto COM (*Component Object Model*) e tornar a sua localização transparente para os utilizadores, possibilitando que uma aplicação cliente possa invocar métodos remotos de modo semelhante ao que acontece com a arquitetura CORBA. Analogamente ao que acontece no COM, o cliente cria um objeto que é capaz de se conectar ao servidor, recebendo uma referência para um *proxy* que representa o servidor, responsável pela comunicação entre o cliente e o servidor.

Apesar do sucesso dos modelos enunciados na integração de *software* em ambiente local, quando surgiu a necessidade de comunicação entre aplicações alojadas em diferentes redes, nenhum dos modelos obteve o sucesso pretendido, apresentando todos eles alguns problemas [Dav07]:

- Possuíam restrições ao nível das plataformas com as quais tinham possibilidade de cooperar;
- Era um grande desafio trabalhar de modo seguro através das *firewalls* da Internet.

No seguimento destes problemas referidos surgiram então os serviços *Web* como uma nova solução para a comunicação entre os sistemas distribuídos e com foco na interoperabilidade entre os diferentes componentes, ambicionando dotar as diferentes aplicações da capacidade de comunicarem entre si, para que o processamento e armazenamento de dados, bem como outras tarefas, possam ser feitas em ambientes distribuídos.

Segundo o *World Wide Web Consortium* (W3C), serviços *Web* podem ser definidos como:

> "métodos standard de interoperabilidade entre diferentes aplicações de software, executando numa variedade de plataformas e / ou frameworks" [BHM+04]

Neste sentido, os serviços *Web* podem ser entendidos como conjuntos de módulos de *software* e dados que estão disponíveis na Internet de modo que outros programas, denominados de clientes, possam comunicar e interagir com os mesmos, enviando e recebendo informações sem qualquer dependência ao nível da linguagem de programação, sistema operativo ou *hardware* utilizado.

Atualmente existem dois tipos principais de arquiteturas para o desenvolvimento de serviços *Web*: SOAP e REST.

2.1.2 SOAP

O *Simple Object Access Protocol* (SOAP) é um protocolo com foco na troca de informações através de ambientes distribuídos e descentralizados [Leo], ou seja, é um protocolo que permite que dois processos que estejam num determinado momento a executar em diferentes ambientes, em máquinas diferentes ou mesmo em redes diferentes, possam comunicar entre si através da Internet.

Através deste protocolo, a comunicação entre os diferentes componentes é realizada com base em mensagens XML, enviadas entre os intervenientes através de mensagens HTTP. Sendo o XML, um formato *standard*, que pode ser interpretado por qualquer tipo de aplicação, independentemente do ambiente e linguagem de programação utilizada, faz com que esta especificação seja transversal, independente da plataforma, simples de usar e robusto.

Para que todos os objetos e métodos disponíveis estejam descritos com clareza, este protocolo define ainda um padrão denominado *Web Service Definition Language* (WSDL), no qual todos os métodos e objetos acessíveis são documentados através de um documento XML, onde está descrita a sintaxe e estrutura dos diferentes pedidos bem com das mensagens de resposta [PL08], facilitando a perceção do serviço por parte dos seus clientes.

O SOAP define ainda um outro padrão, *Universal Description Discovery and Integration* (UDDI), utilizado para o registo de serviços *Web* na Internet para que qualquer cliente possa pesquisar e localizar um determinado serviço, descobrir a especificação necessária para a sua utilização, bem como o tipo de ligações necessárias, o formato das mensagens, etc. Na realidade

este registo não obteve uma aceitação por parte da indústria [PL08], e assim normalmente não existe a entidade UDDI, fazendo com que a comunicação seja direta entre os clientes e os fornecedores de serviços, sendo omitido este ponto intermediário, ou existam registos de WSDL essencialmente ao nível organizacional e de acesso limitado.

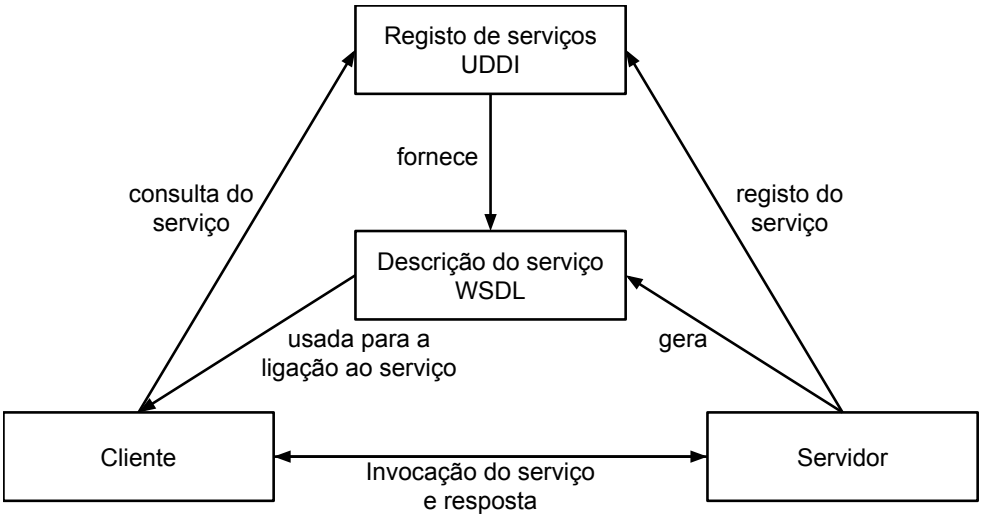

Figura 2.1: Arquitetura de um serviço SOAP

Existem já um leque de motores de SOAP e ferramentas de WSDL que permitem a implementação de um serviço SOAP, de uma forma simplificada, escondendo grande parte da complexidade envolvida no desenvolvimento deste tipo de serviços [PL08], dando a possibilidade de portar, de uma forma simplificada, uma solução existente, que não tenha sido implementada com foco na *Web* para uma implementação orientada à computação remota.

2.1.3 REST

O termo REST foi originalmente proposto por Roy Thomas Fielding, na sua tese de doutoramento intitulada *"Arquitectural styles and the Design of Network-based Software Arquitectures"* [Fie00], na Universidade da Califórnia, Irvine, em 2000, e foi apresentado como um estilo de arquitetura de *software* desenvolvido como foco na criação de serviços *Web*. Esta arquitetura é, muitas vezes, usada como alternativa a outras especificações, como SOAP, CORBA ou RPC.

A arquitetura REST foi definida como "um conjunto coordenado de restrições de arquitetura" [Fie00] que quando aplicadas em conjunto permitem incutir no sistema propriedades tais como usabilidade, simplicidade, escalabilidade e extensibilidade [LC11]. Para a definição deste estilo de arquitetura "híbrido" [Fie00], Roy Fielding partiu de um sistema limpo, sem nenhuma restrição implícita, e, incrementalmente, foi identificando e aplicando restrições aos

elementos do sistema, modificando o estilo de arquitetura, restringindo tanto os papéis e funcionalidades dos diferentes elementos como também as possíveis relações entre os mesmos.

Tal como outros tipos de serviços *Web*, um serviço REST pretende ser independente da plataforma, independente da linguagem, baseado em *standards* já existentes, usando o protocolo HTTP como modo de funcionamento, bem como todas as suas capacidades, desde a possibilidade de utilização de diferentes tipos de métodos, códigos de resposta e cabeçalhos.

Esta arquitetura apesar de não oferecer determinados tipos de funcionalidades na sua base, é flexível o suficiente para poder ser adaptado para que outras funcionalidades como, por exemplo, com segurança e encriptação, possam ser incutidas [Elk08a].

Princípios

Na definição deste estilo de arquitetura foram considerados seis princípios base:

1. Cliente-Servidor

   Separação das diferentes responsabilidades do sistema em diferentes módulos, nomeadamente fazendo a separação entre as componentes de interface do sistema e os módulos responsáveis pelo armazenamento de dados e centralização do serviço.

   Com isto, além da vantagem da separação das responsabilidades num determinado sistema é também possível promover a evolução independente de cada um dos módulos, facilitando a manutenção [CPDM14] e melhorando a portabilidade e a escalabilidade do mesmo [Fie00], desde que as interfaces entre os diferentes componentes sejam bem definidas e mantidas pelas diferentes entidades [Fre13].

2. *Stateless*

   O módulo referente ao servidor do sistema não deve armazenar dados relativos ao estado da aplicação entre os diferentes pedidos. Todas as informações relativas ao estado de uma sessão que sejam necessárias ao processamento dos dados devem ser enviadas pelos clientes em cada um dos pedidos [Fre13]. Apesar de não ser guardado o estado da aplicação, não é invalidada a possibilidade das informações relativas aos diferentes recursos disponíveis, serem armazenadas de modo persistente, já que estas são transversais entre os diferentes clientes [Fre13].

   Este princípio permite uma possível replicação dos servidores, promovendo a disponibilidade, escalabilidade e confiança [CPDM14], no entanto, pode também levar a uma diminuição da performance do sistema devido à necessidade de envio repetido do estado em cada um dos pedidos [Fie00].

3. *Cache*

   Esta propriedade foi adicionada ao estilo de arquitetura com foco na performance do sistema e com o objetivo de evitar interações pedido-resposta quando as informações necessárias já

estão presentes em *cache* [CPDM14], evitando comunicações, reduzindo latência da rede e aumentando a eficiência, a escalabilidade e a perceção de performance [Fie00].

4. Interface Uniforme

   Com foco na simplicidade, acessibilidade, interoperabilidade e na capacidade de descoberta de recursos [CPDM14], esta arquitetura atribui uma grande ênfase a este princípio, sendo uma das caraterísticas principais deste estilo e que mais o diferenciam de outras arquiteturas.

   Este princípio baseia-se nas seguintes restrições [Fre13]:

   - Baseado em recursos: qualquer componente de uma aplicação que possa ser comunicado aos clientes é considerado um recurso e deve ser identificável a partir de um URL único.
   - Manipulação dos recursos através da sua representação: todas as operações realizadas num recurso ocorrem através de trocas de representações desse recurso.
   - Mensagens auto-descritivas: todas as mensagens devem conter as informações necessárias para o devido tratamento pelo destinatário.
   - *Hypermedia as the Engine of Application State* (HATEOAS): uma determinada API REST deve ser navegável, sendo possível, a partir de um determinado recurso, descobrir quais os recursos que lhe estão associados e ainda ser possível encontrar as informações relativas aos mesmos.

5. Sistema em Camadas

   A utilização de diferentes camadas no desenho do sistema permite encapsular e proteger determinadas componentes do serviço, fazendo com que nem todas sejam acessíveis de modo público, obrigando deste modo à utilização de componentes intermediários para a gestão de acessos. Através da utilização de componentes intermediários é possível fazer um melhor balanceamento do sistema, com foco na disponibilidade, na escalabilidade e ainda na segurança [CPDM14].

6. *Code-On-Demand*

   Este princípio tem como objetivo dotar o servidor da capacidade de transferência de código que pode ser utilizado no cliente, estendendo as suas funcionalidades, e permitindo simplificação da sua implementação [Fie00]. Este princípio promove a extensibilidade do sistema, mas, no entanto, a complexidade do cliente é aumentada, já que este necessita de lidar com adições de funcionalidades em tempo de execução. A interoperabilidade do sistema é também afetada, dado que o código transferido deve ser compatível com a implementação do cliente [CPDM14].

Boas Práticas

Além dos princípios definidos por Roy Fielding para a arquitetura REST, para que os serviços REST apresentem uma maior qualidade e uma maior acessibilidade para os seus clientes, foram ainda definidos um conjunto de conceitos e boas práticas a ter em conta nas implementações [Fre13]:

- Utilização correta do protocolo HTTP na manipulação dos recursos disponíveis a partir de um endereço, desde a utilização correta dos métodos HTTP (GET, POST, PUT, DELETE, OPTIONS, PATCH, HEAD), a definição apropriada dos cabeçalhos dos pedidos bem como a correta codificação das respostas, aproveitando os significados *standard* dos diferentes códigos de resposta (ver anexo B).

- Definição dos nomes dos recursos, URIs, de uma forma cuidada, de forma a melhorar a compreensão do sistema por parte dos seus clientes. Assim, todas as operações sobre o mesmo recurso devem usar o mesmo nome, os recursos devem ser identificados pelo URL e não pelos parâmetros do pedido, e ainda, o nome dos recursos deve estar no plural, por forma a dar a entender que dá acesso à coleção dos recursos e não apenas a um único recurso.

- Como tipos de formatos de renderização dos dados devem estar disponíveis os formatos JSON e XML, devendo ser usado como predefinição o primeiro formato enunciado, e, a menos que os custos de implementação de dois serializadores seja acrescido, devem ser também oferecida a possibilidade de representação no formato XML, aumentando a interoperabilidade com outros sistemas [Fre13].

- Preferência na criação de recursos de baixa granularidade, dotando-os de todas as operações CRUD, podendo partir dos mesmos para a construção de recursos mais elaborados, agregando outros já existentes como forma de diminuir a quantidade necessária de comunicações.

- Como forma de manter a compatibilidade com os clientes já existentes, as alterações no serviço devem levar à criação de novas versões, permitindo os clientes continuarem a usar os serviços até que eles próprios façam a adaptação à nova versão [Jam12].

- Desenvolvimento e manutenção da documentação do serviço, de preferência com exemplos de utilização em várias linguagens de programação, de modo que a aprendizagem do serviço por parte dos clientes seja o mais simples possível.

Estas são as restrições/princípios descritos por Roy Fielding que representam uma arquitetura REST "pura" [CPDM14], no entanto, esta arquitetura não é "standard" [Elk08b], é apenas um conceito e um conjunto de princípios e boas práticas, não existindo requisitos para que um determinado serviço deva funcionar de uma forma particular [Jam12].

Após a descrição dos dois tipos de arquitetura, SOAP e REST, no anexo A pode-se encontrar uma análise comparativa dos mesmos atendendo a aspetos como o formato das mensagens, o protocolo de transporte, design, descrição do serviço e segurança.

2.2 JavaScript

JavaScript é uma linguagem de programação interpretada, multi-plataforma e orientada a objetos, projetada em 1995 durante apenas 10 dias [SE12], por Bredan Eich enquanto trabalhava para a empresa Netscape Communications Corporation, sendo por isso usada em primeiro lugar nos *browsers* Nestscape. Esta linguagem foi desenhada com foco na extensão das funcionalidades das páginas *Web* através da execução de código executável do lado do cliente diretamente no *browser*, tornando possível a implementação de páginas *Web* mais interativas, permitindo interação com o utilizador, comunicação assíncrona com outras aplicações *Web* bem como alterar o conteúdo apresentado.

JavaScript é uma linguagem baseada em protótipos [RLBV10], com capacidade para vários paradigmas, possuindo suporte para programação através de diferentes estilos, tais como imperativo, orientado a objetos [RLBV10] e funcional [TV10].

De modo que outros fabricantes de *browsers* pudessem seguir os trabalhos feitos por Netscape, em 1996-1997, a linguagem JavaScript foi levada para *European Computer Manufacturers Association* (ECMA) de modo a ser criada uma especificação padrão com base na linguagem JavaScript, que posteriormente ficou denominada de ECMA-262, ficando o padrão oficial conhecido como ECMAScript [W3C12]. Assim, ECMAScript pode ser definido simplesmente como uma descrição, definindo todas as propriedades, métodos e objeto de uma linguagem de script [Zak09]. Com isto, e obedecendo à especificação definida, surgiram algumas implementações como é o caso de JavaScript, sendo a implementação mais conhecida, existem também outras como JScript e ActionScript [RLBV10].

Desde então, JavaScript foi-se tornando uma das linguagens de programação mais populares e de eleição para o desenvolvimento na *Web*, tendo a sua especificação e *standards* sido acompanhada de diversas evoluções, tendo já sido lançadas 5 edições da especificação ECMA-262, estando neste momento em desenvolvimento a 6ª edição, com nome de código *Harmony*. Além disso, existe também já tanto proposta como algum trabalho desenvolvido relativamente à 7ª edição desta especificação.

Baseada em protótipos

Ao contrário do que acontece com outras linguagens de programação orientadas a objetos como C++, C# ou Java, a linguagem JavaScript, apesar de também ser orientada a objetos, não é baseada em classes. Com isto, em vez de a herança entre objetos se processar através do método clássico em que são definidas classes que podem herdar propriedades de outras classes, em JavaScript os objetos podem herdar propriedades diretamente a partir de outros objetos,

funcionando os objetos base como protótipos [Cro], sendo este um dos principais fatores que diferenciam esta linguagem de outras.

Neste sentido, todos os objetos em JavaScript possuem uma propriedade denominada *prototype*, que permite uma ligação interna do mesmo a outro objeto. Assim, quando existe herança entre objetos, este contém o protótipo do seu objeto pai, e este pode ter também ligação para outro objeto, formando assim uma cadeia de protótipos [RLBV10].

Closures

Closures são um dos recursos mais poderosos do ECMAScript, permitem que qualquer função definida internamente relativamente a outra função, tenham acesso a todas as variáveis locais, parâmetros, e outros métodos definidos também internamente. Deste modo, uma *closure* é formado quando uma função interna é acessível do lado de fora da função em que foi declarada, permitindo a sua execução, acedendo a todas as variáveis e parâmetros internamente definidos, mesmo que a execução da função exterior tenha terminado.

Assim, qualquer função definida internamente tem acesso não só ao seu contexto de execução, mas também ao contexto de execução da função exterior bem como ao contexto global, podendo também aceder às variáveis globais.

2.2.1 Ecmascript 6

Relativamente à evolução da especificação e da linguagem, a 6ª edição, com o nome de código *Harmony*, encontra-se em fase de lançamento e estará disponível ainda em 2015.

Esta edição será a responsável por alterações na linguagem bem como a introdução de novos conceitos. Do conjunto de conceitos introduzidos, alguns deles destacam-se sobretudo pela mudança que os mesmos introduzem na linguagem. Assim, esta nova edição é responsável pela introdução de conceitos como classes, *arrows*, *template strings*, *let & const*, *generators*, *modules*, *promises*, entre outros, bem como a melhoria de alguns conceitos já existentes.

Classes

Apesar de já ser possível fazer implementação de classes através de uma sintaxe baseada em protótipos, o processo era de certo modo verboso. Deste modo, embora sejam igualmente baseadas em protótipos, as classes ES6[1] oferecem ao programador uma forma mais declarativa e fácil de usar, suportando tanto a herança no modo de protótipos, bem como chamadas a super classes, instanciação de objetos, definição de métodos estáticos e construtores.

Arrows

Arrows definem uma nova sintaxe de definição de funções, semelhante ao que já existe em linguagens como C#, Java 8 ou CoffeeScript, que permite declarar uma função de uma forma

[1] EcmaScript 6

mais abreviada. Apesar de o comportamento ser semelhante à utilização de funções, as *arrows* partilham o *this* com o código envolvente.

Módulos

Apesar de existirem já soluções para a utilização de módulos em JavaScript, nenhuma das soluções já existentes desenvolvidas pela comunidade possuía suporte nativo pela linguagem. É neste sentido que surgem então os módulos ES6, dotando a linguagem de suporte nativo para a definição e utilização de módulos, indo ao encontro de algumas das soluções mais populares já existentes (AMD[2], CommonJS).

Generators

O JavaScript além de possuir funcionalidades que lhe permite receber o resultado de uma operação de modo assíncrono, possui também forma de permitir suspender a execução de uma determinada operação através de *generators*.

Generators são funções que podem ser suspensas e posteriormente novamente colocadas em execução, sendo o seu contexto de execução mantido [Net14a].

Devido às suas caraterísticas, os *generators* são consideradas uma alternativa à implementação de iteradores, dado que quando um determinado gerador é invocado, o corpo da função não é executado, retornando em vez disso um objeto gerador-iterador, através do qual é possível executar as várias iterações individualmente.

Promises

Uma das caraterísticas desta linguagem é o facto de ser *single-threaded*, impossibilitando assim a execução de várias tarefas ao mesmo tempo, sendo necessário que a execução das diferentes tarefas seja sequencial. Além disso, o JavaScript ainda partilha a *thread* com outros processos que ocorrem no *browser* tais como o desenho das interfaces, a atualização dos estilos e tratamento de eventos, sendo que cada uma dessas atividades causa atraso na execução das restantes [Arc14].

Uma das soluções encontradas para que haja execução de tarefas de modo não bloqueante é o registo de funções de *callback* que serão executadas após a execução da operação em questão, evitando assim o bloqueio da sequência das operações. No entanto, esta abordagem não garante a sequência das operações, e em casos em que há dependência de dados é necessário fazer o que chamam de *"Pyramid of Doom"*, em que são colocados invocações de operações assíncronas dentro de *callback* de operações assíncronas anteriores, garantindo a sequência das operações mas diminuindo muito a legibilidade do código em questão.

Para a resolução do problema mencionado foi introduzida no JavaScript a abordagem de promessas, ou *promises*, que funciona com um objeto *proxy* que representa um resultado desconhecido que ainda não foi computado [KBM13]. Deste modo, uma *promise* associa às

[2] Asynchronous Module Definition

operações assíncronas *handlers* de sucesso e de falha que, posteriormente, serão utilizados de modo a retornar os valores resultantes de uma determinada operação, semelhante ao que acontece com as operações síncronas.

Concluindo, uma operação assíncrona devolve uma promessa que é a responsável por retornar o valor resultante da operação num ponto futuro [Net14b].

2.2.2 EcmaScript 7

Apesar da 6ª edição da especificação EcmaScript estar ainda em fase de conclusão e lançamento existe já uma nova proposta referente à 7ª edição da mesma. Esta nova proposta pretende dotar a especificação de um conjunto de caraterísticas que vêm melhorar a utilização de alguns dos conceitos introduzidos na versão anterior, destacando-se as seguintes, de acordo com o desenvolvimento do projeto em foco nesta dissertação:

- **Propriedades das classes**: Na sexta edição da especificação do JavaScript foi introduzido o conceito de classes, que permite 'mascarar' a utilização de protótipos, no entanto pormenores como a definição de propriedades da classe, tanto estáticas como de instância não são possíveis através da sintaxe definida no ES6. Neste sentido, esta nova proposta de especificação pretende que esse processo seja disponível diretamente na sintaxe das classes, e ao mesmo tempo permitindo que essas propriedades sejam associadas às instâncias da classe.

- **Funções assíncronas**: A introdução de *promises* e *generators* na 6ª edição abriu portas a esta nova funcionalidade. Tal como nos *generators* é possível esperar pelo resultado de uma operação assíncronas implementadas através de *promises* também esse comportamento deveria ser possível na utilização de funções. No ES6 isto é possível através das *promises*, no entanto, apesar de a utilização ter sido melhorada em relação à utilização de *callbacks*, a forma como os programadores terão de programar a utilização de *promises* pode ser igualmente desafiante. Neste sentido, tal como nos *geradores*, esta funcionalidade permitirá declarar uma função como assíncrona e indicar que e pretende esperar pelo resultado das operações assíncronas nela incluídas, de uma forma programaticamente mais simples e apelativa.

- ***Decorators***: Esta funcionalidade é um dos pontos mais interessantes existentes nesta proposta para o ES7[3]. O conceito de *decorators*, de modo semelhante ao que já existe em linguagens como Java e Python, são expressões que se podem associar às classes ou aos seus métodos e que permite que seja executadas tarefas sobre as mesmas e ao mesmo tempo manter a sintaxe declarativa das classes ES6. Os *decorators* são representados por funções e permitem anotar a classe, durante a definição da mesma, ou ainda fazer alterações às diferentes propriedades da mesma, modificando assim o seu comportamento.

[3] EcmaScript 7

2.2.3 Node.js

O aparecimento de projetos como Node.js permitiram tornar o JavaScript uma linguagem possível de ser usado na implementação de componentes de um servidor de uma aplicação *Web* [SE12]. Deste modo, sendo o JavaScript uma das linguagens mais utilizadas na programação de aplicações *Web* do lado do cliente, a possibilidade de implementação de servidor através de Node.js permite assim uma maior potencial de integração entre os cliente e o servidor [CKT14].

Node.js é uma plataforma de desenvolvimento de aplicações *Web*, desenvolvida utilizando o *JavaScript Engine V8* do Google, com o objetivo de proporcionar uma forma fácil de construir aplicações rápidas e escaláveis [Jun12]. Trata-se de um ambiente de desenvolvimento de JavaScript, desenvolvido por Ryan Dahl em 2009, maioritariamente implementada em C e C++, focando a performance e o baixo consumo de memória e visando suportar processos de servidor de longa duração [TV10].

Ao contrário do que acontece com outros ambientes de desenvolvimento, o Node.js não depende de *multithreading* para a execução de processos concorrentes na lógica de negócio do servidor. Na realidade, deve-se entender o Node.js como uma plataforma de desenvolvimento de servidores *Web* que possibilita a construção de sistemas altamente escaláveis, sem as complexidades de gestão de um sistema *multithreading* [SE12], operando apenas em *single-thread*, através de uma abordagem baseada em eventos e de *input* e *output* não bloqueante.

Deste modo, esta tecnologia utiliza operações de I/O não bloqueantes, assíncronas ou orientadas a eventos, simplesmente registando uma função de *callback* que é invocada quando a operação em causa é concluída, prevenindo assim que a aplicação não fique bloqueada aquando da execução deste tipo de operações. Com isto é assim possível mudar o paradigma tradicional, no qual a maioria das operações de I/O são bloqueantes, suspendendo a execução do programa principal até que as operações sejam concluídas [Sch14], para uma abordagem onde todas as operações podem ser executadas de forma assíncrona, não bloqueante, criando todo um sistema como base num modelo de eventos.

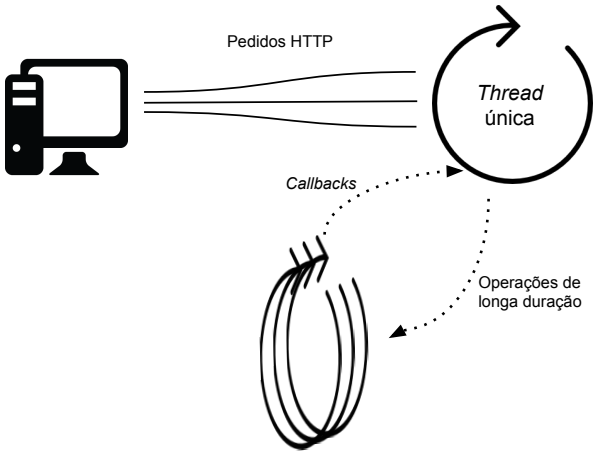

Figura 2.2: Processamento das operações em Node.js

Segundo Abernethy "o Node.js é extremamente bem projetado para situações em que um grande volume de tráfego é esperado e o processamento necessário do lado do servidor, antes de responder ao cliente, não é volumoso" [Abe11].

Outro ponto a ter em consideração, é a utilização da capacidade de processamento de um sistema operativo na integra, ou seja, a utilização de vários processadores ou núcleos disponíveis. Como o Node.js executa num único processo, uma das soluções possíveis e já utilizada é a execução de múltiplas instâncias do mesmo processo, e a posterior integração de um módulo desenvolvido pela comunidade, *multi-node*, que permite utilizar as capacidades dos sistemas operativos, compartilhando *sockets* entre os diferentes processos [Jun12].

Por fim, através do *Node Package Manager* é possível fazer a integração de um vasto conjunto de módulos e pacotes externos desenvolvidos por uma grande comunidade de desenvolvimento e de apoio e que os disponibiliza de modo público, para que qualquer programador possa partilhar ou reutilizar código de terceiros de uma forma simples. Assim é possível fazer a implementação de uma aplicação composta por um conjunto de pequenos módulos, que são mantidos pelos seus criadores.

2.3 REST *Frameworks*

A utilização do estilo de arquitetura REST para a construção de serviços *Web* tornou-se cada vez mais popular e com isso foram propostas e desenvolvidas diversas *frameworks* capazes de providenciar aos programadores uma forma mais simples e acessível de implementar aplicações seguindo um arquitetura REST.

Para o estudo das soluções já existentes no mercado, foi escolhido um conjunto de *frameworks* que atuassem através de linguagens de programação diversificadas. Em primeiro lugar serão apresentadas algumas *frameworks* destinadas à implementação de serviços REST nas linguagens Python, Java e Ruby, tendo sido escolhidas as soluções mais relevantes e utilizadas pela comunidade. Por último, são apresentadas soluções para o desenvolvimento deste tipo de serviços destinadas à linguagem JavaScript, sendo dado um maior foco do que as anteriores, dado que esta dissertação tem como alvo o desenvolvimento nesta linguagem.

2.3.1 Django REST Framework

Django REST Framework é uma ferramenta considerada "poderosa, sofisticada e surpreendentemente fácil de usar" [Mat14], que pode ser usada em conjunto com a *framework* de desenvolvimento de aplicações *Web* Django, que quando integrada no desenvolvimento de um determinado *backend* permite a implementação de serviços REST.

Esta *framework* é dotada de um conjunto de funcionalidades que permite estender as caraterísticas das APIs REST de uma forma simples e intuitiva. Um dos aspetos mais diferenciadores é a possibilidade de disponibilização de uma interface *Web* navegável da API,

oferecendo uma grande usabilidade e simplicidade, bem como uma compreensão e a descoberta da mesma facilitada por parte dos utilizadores.

Um dos pontos fortes desta *framework* é a capacidade de construção do serviço através de vistas baseadas em classes, permitindo de um modo quase automático fazer a construção da API através dos modelos dos recursos disponíveis, não sendo necessário para implementar a lógica de cada um dos *endpoints*[4] disponibilizados, havendo no entanto a possibilidade de personalização da lógica de negócio de cada uma das rotas.

O Django REST Framework oferece uma poderosa capacidade de manipulação do modo como os recursos são apresentados através do uso de serializadores, permitindo haver negociação do formato em que os recursos são representados, mas também a personalização de serializadores próprios por forma a satisfazer as necessidades do serviço, a capacidade de autenticação de utilizadores, controlo de acessos através de permissões e ainda capacidade de teste da API [Hol14].

Por fim, oferece a possibilidade de utilização de boas práticas da arquitetura REST de um modo simples, tais como paginação de conteúdos, utilização de filtros de conteúdos, negociação de formatos de dados, entre outras. Além disso, esta *framework* disponibiliza uma documentação bastante detalhada, explicativa e uma grande comunidade de apoio que faz com que seja mantido o suporte e o desenvolvimento desta solução.

2.3.2 Flask-RESTful

Flask-RESTful é uma extensão para *microframework* Flask, simples e fácil de usar para o desenvolvimento de serviços simples através da linguagem de programação Python.

Esta solução oferece aos programadores uma interface limpa para a fácil interpretação dos diferentes argumentos dos recursos, formatação/serialização de respostas e organização das diferentes rotas disponíveis. O Flask-RESTful permite abstrair as tarefas relacionadas com o protocolo HTTP, no entanto é completamente independente da lógica de negócio da aplicação, permitindo que os programadores tenham liberdade para implementar o serviço tendo em conta as suas necessidades e preferências [Hor12].

O principal objetivo desta *framework* é oferecer aos programadores um ponto de partida simples e extensível para a implementação das suas APIs, sem a necessidade de inclusão de outras dependências além do Flask.

Deste modo, esta é uma *framework* ideal para o desenvolvimento de serviços simples, sem necessidade de configurações acrescidas de ORM, autenticação, administração, etc., tornando-se deste modo uma solução para o desenvolvimento rápido, mas ao mesmo tempo extensível.

2.3.3 Restlet

Restlet é uma *framework open source* para o desenvolvimento de APIs REST na linguagem de programação Java, disponível tanto para a implementação de servidores como também para a

[4] Endereço ou ponto de conexão a um serviço *web*

implementações de clientes REST, através da mesma API, reduzindo assim a curva de aprendizagem dos diferentes componentes do sistema.

Esta ferramenta faz a aproximação dos conceitos básicos de uma arquitetura REST, incluindo recursos, representações, conectores e componentes, a artefactos equivalentes em Java.

Através desta *framework*, a distribuição do serviço pode também tornar-se flexível dando a possibilidade de incluir o módulo relativo à interface REST numa outra aplicação Java já existente ou ainda fazer disponibilizar apenas como uma aplicação de servidor que pode ser integrada num servidor HTTP. Assim, esta *framework* está disponível para diversos ambientes de desenvolvimento, tais como, Google Web Toolkit, Google App Engine, Android, Java SE/EE, e OSGi.

Esta solução possui ainda a vantagem de manter um mecanismo de extensões que permite incluir bibliotecas de Java externas de modo a suportar características relacionadas com aspetos como bases de dados, segurança, *templates*, e até outros tipos de serialização de conteúdos.

O estabelecimento de restrições de acesso aos recursos pode também ser feito através desta ferramenta, possuindo suporte para autenticação através de diversos métodos como autenticação básica HTTP, Amazon S3, OAuth2.0. e ainda suporte para HTTP, SMTP e POP através de SSL.

2.3.4 Spark

Spark é uma *microframework* para o desenvolvimento de serviços REST na linguagem Java. Esta solução tem como principal objetivo proporcionar aos programadores o desenvolvimento rápido deste tipo de serviços *Web*, de uma forma simples, leve e com um esforço reduzido.

Ao contrário do que acontece com grande parte das *frameworks* de desenvolvimento de serviços REST em Java, esta tenta evitar a necessidade de utilização de uma grande quantidade de configurações que, normalmente, as tornam bastante verbosas, fugindo assim ao paradigma de desenvolvimento com recurso ao *standard* JAX-RS, tentando proporcionar um desenvolvimento através de Java puro.

Esta *framework* é bastante minimalista, providenciando as funcionalidades mínimas necessárias à implementação deste tipo de serviços, nas quais estão incluídas a gestão das diferentes rotas disponíveis, manipulação de pedidos HTTP e das respetivas respostas, gestão de dados de sessão, *cookies*, e ainda a capacidade de renderização de *templates Java Server Pages* (JSP).

2.3.5 Sinatra

Sinatra é uma *framework* gratuita e *open source* desenvolvida na linguagem de programação Ruby, com o objetivo de permitir a implementação de serviços REST de uma forma rápida, acessível e ao mesmo tempo elegante, leve e minimalista. Ao contrário do que acontece com outras *frameworks* deste tipo, esta não usa o padrão *model-view-controller*, tentando ser o mais simples e flexível possível, proporcionando aos programadores uma forma de implementação de serviços com o menor esforço possível.

Apesar de simples, é dotada de um conjunto de funcionalidades que permite ao programador adaptar o serviço às suas necessidades. As principais funcionalidades centram-se na capacidade de gestão das diferentes rotas, bem como as diferentes respostas possíveis, renderização de *templates* num vasto conjunto de formatos, gestão de ficheiros estáticos, sessões, *logging*, controlo de *cache* e outras funcionalidades de apoio ao serviço, algumas das quais suportadas pela adição de *middlewares* ao serviço.

Esta solução apesar de não ser a mais usada na linguagem de programação alvo, para o desenvolvimento de APIs ou de aplicações de pequenas dimensões, sobre as quais é pretendido ter um controlo de todos os aspetos, é considerada pela comunidade como uma boa solução, permitindo o desenvolvimento de serviços de um modo elegante e minimalista [Jon12].

2.3.6 Express

Express é uma *micro-framework* minimalista, flexível, desenvolvida em JavaScript e inspirada em Sinatra, que permite desenvolver serviços REST, mas também outros tipos de aplicações *Web*, através de sistemas de renderização de *templates* e gestão de rotas disponíveis.

Para o desenvolvimento de APIs REST, esta *framework* possui um conjunto de recursos e funcionalidades que permite aos programadores, de um modo simples e rápido, fazer a criação do esqueleto referente ao serviço REST pretendido, sendo necessário ao programador desenvolver a lógica de negócio referente aos diferentes *endpoints* disponíveis.

Assim, esta *framework* é aconselhada principalmente para casos em que as aplicações *Web* contém muitas funcionalidades além do serviço REST , onde a lógica de negócio é relativamente complexa, haja necessidade de utilização de *middlewares* ou, por outro lado, haja a necessidade de renderização de *templates*. Esta solução não é aconselhada a serviços onde estas funcionalidades não sejam usadas e a implementação da lógica da aplicação originaria a uma elevada repetição do código para a manipulação dos diferentes recursos.

2.3.7 Restify

Restify é uma *framework* em Node.js construída especialmente para a correta implementação de serviços REST, utilizando, de um modo intencional, muitas das caraterísticas da *framework* Express, descrita na secção anterior.

Tal como acontece no Express, com esta *framework* existe a necessidade de implementação de cada uma das rotas do serviço REST, sendo o programador obrigado a tratar de toda a lógica inerente a cada um dos *endpoints* de acesso aos recursos levando a um grande esforço de implementação muitas das vezes repetitivo.

Em relação ao Express, esta solução tem como principais vantagens o facto de ser diretamente focado em implementações de serviços REST, possuindo apenas funcionalidades relacionadas com este tipo de serviços. Esta solução não inclui funcionalidades relacionadas com *templates* e renderização, possuindo por sua vez características propícias à gestão deste tipo serviços como

controlo de *cache*, controlo de acessos, filtro de recursos, *logging* de acessos e ainda utilização de ligação entre os recursos por forma a tornar o serviço navegável.

2.3.8 SailsJS

SailsJS é uma *framework* para o desenvolvimento de serviços REST com base no padrão *model-view-controller*, implementada também com base em Express.

Esta solução permite a implementação de *backends*, de uma forma rápida, através de uma abordagem de geração automática de código, permitindo aos programadores desenvolverem os seus serviços através de uma interface de linha de comandos. Assim, os programadores podem gerir os modelos de dados, as operações CRUD associadas aos mesmos, bem como as operações de ordenação, filtros, pesquisas, paginações e associações de um modo relativamente simples.

Relativamente à persistência de dados, esta solução possui uma implementação de ORM, o Waterline, compatível com diferentes tipos de bases de dados, tais como MySQL, MongoDB, PostgreSQL, Redis, entre outras, permitindo assim que o serviço seja independente do tipo de fonte de dados em utilização.

Esta *framework* permite ainda o desenvolvimento de sistemas direcionados para uma abordagem *real-time* sem esforço de adição de código para este propósito, permitindo a utilização de *sockets*[5] cujas mensagens são convertidas tornando-se compatíveis com as rotas da API.

Relativamente a políticas de segurança, esta solução disponibiliza funcionalidades de controlo de acessos, através de autenticação e autorização nos pedidos enviados ao serviço.

2.3.9 LoopBack

Loopback é uma *framework open source*, criada por StrongLoop, desenvolvida com foco na criação dinâmica de APIs REST com um mínimo de esforço de implementação.

A criação de serviços é facilitada pela existência de duas ferramentas que ajudam a criar e a gerir aplicações sem a necessidade de implementação de código de servidor. A ferramenta *slc loopback* permite que, através de uma interface de linha de comandos, seja feita a gestão do serviço, desde a criação da aplicação até à gestão dos modelos de dados e as respetivas propriedades, relações entre eles e gerir as fontes de dados. Existe ainda uma outra ferramenta, StrongLoop Arc, que também permite fazer a gestão do serviço, mas desta vez, através de uma interface gráfica, mas para além destas funcionalidades permite ainda ter acesso a métricas de performance do serviço, recolher dados relativos à utilização de memória e de CPU, e ainda fazer a distribuição do respetivo serviço.

Loopback possui já pré-definidos um conjunto de modelos de dados, responsáveis pela entidade utilizadores, de modo a facilitar as tarefas de gestão de utilizadores, bem como os processos a eles associados. Permite ainda a utilização de mecanismos de autorização, através da associação de utilizadores a diferentes tipos de papéis, podendo assim restringir o acesso de

[5]Mecanismo em que através da criação de um canal é possibilitada a comunicação entre várias aplicações

determinadas funcionalidades a apenas alguns utilizadores tendo em conta os seus papéis no sistema.

Esta solução para o desenvolvimento de serviços REST permite ainda o envio de notificações *Push* para diferentes tipos de dispositivos móveis e a ligação a serviços externos tanto para o *upload* e *download* de ficheiros de serviços como para a autenticação de clientes.

2.3.10 Comparação de *frameworks*

Após o estudo e a descrição das diferentes *frameworks* escolhidas, foi elaborada uma tabela comparativa (ver anexo C) na qual estão presentes as principais caraterísticas relativas a soluções desta área de intervenção, tentando salientar os aspetos que podem ser comuns nas diferentes linguagens, de modo a que a análise possa ser o mais transversal possível.

Tabela 2.1: Tabela comparativa de *frameworks* REST resumida

	Linguagem	**Baseada em Recursos**	**Cache**	**HATEOAS**
Django Rest Framework	Python	Sim	Sim	Sim
Flask-RESTful	Python	Sim	Não	Não
Restlet	Java	Não	Não	Não
Spark	Java	Não	Não	Não
Sinatra	Ruby	Não	Sim	Não
Express	JavaScript	Não	Não	Não
Restify	JavaScript	Não	Sim	Não
Sails	JavaScript	Sim	Não	Não
Loopback	JavaScript	Sim	Não	Não

De entre as *frameworks* de Python analisadas, a Django REST Framework é aquela que apresenta um grau de maturidade maior, apresentando um grande conjunto de funcionalidades aos programadores, permitindo a utilização de boas práticas de desenvolvimento de serviços REST, aproveitando todas as vantagens da plataforma Django, nomeadamente o ORM definido pela própria plataforma, autenticação, gestão de base de dados, de rotas, entre outras. Por outro lado, a *Flask-RESTful* é mais simples e minimalista, permitindo desenvolver serviços REST de uma forma rápida e leve, possuindo apenas as funcionalidades mais básicas necessárias para o desenvolvimento deste tipo de serviços. Deste modo, para o desenvolvimento de serviços mais complexos o Django REST Framework é uma boa solução e bastante completa, no entanto, para o desenvolvimento de serviços mais simples pode tornar-se uma *framework* mais pesada e complexa, sendo soluções como a Flask mais indicadas para a implementação de APIs relativamente simples.

Relativamente às soluções analisadas, na generalidade, aquelas cujo alvo é o desenvolvimento em Java, são as que obrigam a um maior esforço de configuração e de implementação, muito devido à necessidade de cumprimento das normas do *standard* JAX-RS, obrigando a configurações acrescidas, muitas vezes através de ficheiros XML, levando assim a uma maior fragmentação da

implementação dos serviços. No entanto, existem *frameworks* mais simples, como é o caso do Spark, que permite o desenvolvimento de serviços tentando resolver os problemas de verbosidade das restantes analisadas, mas que ao mesmo tempo o grau de maturidade é mais reduzido e cujo nível de funcionalidades que disponibiliza aos programadores é mais minimalista.

Relativamente à única *framework* de Ruby analisada, a Sinatra, em comparação com grande parte das outras soluções estudadas, é bastante simples e minimalista, mas ao mesmo tempo elegante. Ao contrário do que acontece com outras, esta não usa uma abordagem *model-view-controller*, assentando apenas na definição e implementação individual de cada um dos métodos disponíveis. O estilo de desenvolvimento que esta *framework* propicia, serviu de base a outras soluções da área como é o caso de Express.

No que diz respeito às *frameworks* para o desenvolvimento de serviços REST em Node.js, foram analisadas soluções cujo paradigma de desenvolvimento difere entre si.

Inicialmente foram analisadas as *frameworks* Express e Restify, nas quais, o desenvolvimento dos serviços assenta na implementação de cada um dos métodos disponíveis para a manipulação dos diferentes recursos, tal como acontece com a *framework* Sinatra. As duas soluções são semelhantes, tanto que a Restify é baseada em Express, no entanto a Express é uma solução para o desenvolvimento de aplicações *Web* no geral, independentemente de se tratar ou não de uma API REST, enquanto a Restify tem como único foco a implementação de serviços REST, sendo as suas funcionalidades relacionadas diretamente com este tipo de serviços, tais como gestão de *cache*, controlo de acessos, filtros e *logging*.

Foram ainda analisadas outras duas soluções em Node.js, SailsJS e Loopback, que propiciam o desenvolvimento de serviços REST, com um reduzido esforço de implementação através de código. Estas duas soluções permitem a criação e a gestão do serviço REST, através de uma interface de linha de comandos, permitindo a criação dos modelos dos recursos, bem como a gestão das suas diferentes propriedades, associações, ligações a fontes de dados através da utilização de ORMs bem definidos, possibilitando a utilização de diferentes fontes de dados de modo praticamente transparente. A *framework* SailsJS, em comparação com a Loopback, é uma solução mais simples, permitindo fazer a implementação de APIs de uma forma rápida e com foco numa abordagem *real-time*. Por outro lado, a *framework* Loopback possui também uma interface gráfica, através de StrongLoop Arc, que é uma das suas principais vantagens e aspetos inovadores, permitindo fazer tanto a gestão da lógica do serviço, dos utilizadores que podem aceder ao mesmo e os seus papéis dentro do sistema e ainda controlar outros tipos de propriedades do sistema como a utilização de memória e de CPU, de um modo mais apelativo, sem necessidade de implementação de código de servidor.

Após análise de algumas das diferentes soluções existentes no mercado, foi possível perceber a *framework* indicada depende da complexidade do serviço a desenvolver. Assim, para serviços mais complexos existem soluções mais completas como Django REST Framework, SailsJS ou Loopback, enquanto para o desenvolvimento de APIs mais simples, estas soluções poderão ser pesadas e obrigar a um esforço de implementação maior, e por isso, soluções como Restify, Flask ou Spark poderão ser as ideais.

2.4 Documentação Automática

Neste momento, a arquitetura REST é a mais utilizada para a implementação de serviços *Web*. No entanto, e como já referido anteriormente, esta arquitetura não define *standards* para a descrição e documentação das especificações de um determinado serviço REST, sendo que atualmente grande parte das APIs são documentadas manualmente, e, portanto, é difícil de manter a documentação perfeitamente sincronizada com a API [Cog13].

De modo a ultrapassar este ponto fraco da arquitetura REST, foram surgindo alguns métodos e ferramentas para a descrição de APIs REST como, por exemplo, Swagger, API Blueprint, I/O Docs ou RAML, que pretendem dotar os serviços REST de uma forma de descrição que possa ser intuitiva, simples de utilizar, humanamente interpretável e em alguns dos casos, capaz de ser analisada por computadores.

2.4.1 Swagger

É uma especificação para descrever, produzir, consumir e visualizar serviços RESTful. Tem como principal objetivo proporcionar uma forma fácil de documentação de uma determinada API, possibilitando que a mesma seja atualizada paralelamente ao desenvolvimento do respetivo serviço, através da integração da estrutura de documentação no respetivo código do serviço.

Este tipo de solução proporciona aos serviços desenvolvidos, através de uma arquitetura REST, que à partida não possui uma especificação para documentação e descrição, uma forma de documentação que permite ainda que qualquer utilizador possa perceber e utilizar o serviço sem ter conhecimento da respetiva implementação nem acesso ao código do serviço [Rev].

O Swagger possui ainda uma componente interativa onde, para além de ser disponibilizada a documentação do serviço, qualquer pessoa pode utilizar os formulários disponibilizados como forma de aceder à API, interagir com a mesma e ainda perceber quais os tipos de respostas que cada um dos métodos poderá disponibilizar.

2.4.2 Slate

Slate é uma *framework* destinada a ajudar na criação de documentação para APIs, funcionando como um *template* inteligente e responsivo para a documentação de serviços REST.

Esta ferramenta permite que toda a documentação seja escrita usando apenas Markdown, tornando-a simples de perceber e de editar. Após a escrita da documentação, esta é apresentada ao utilizador como uma página *Web* utilizando um conceito de *single page*, sendo todas as informações apresentadas na mesma página, de uma forma limpa e intuitiva, evitando a navegação por um vasto conjunto de páginas para ter acesso às informações pretendidas.

Por predefinição a documentação é guardada num repositório público de GitHub, permitindo que qualquer utilizador tenha acesso à fonte da documentação e ainda contribuir com melhorias para a mesma.

2.4.3 API Blueprint

É uma linguagem de descrição orientada à documentação de APIs *Web*, constituída basicamente por um conjunto de suposições feitas através da sintaxe Markdown.

A documentação do serviço em questão é produzida através da escrita de um documento obedecendo à Markdown, no qual o serviço *Web* ou apenas parte dele é descrito, sendo os criadores da API os responsáveis por escolher quais as partes que pretendem documentar.

A documentação produzida tem de obedecer a uma determinada estrutura, dado que cada uma das secções possui um significado e uma posição diferentes no documento, tendo em conta a estrutura base da especificação. No entanto, não é obrigatório que o documento em questão possua todas as secções, sendo que todas elas são opcionais, embora que, no caso das mesmas existirem necessitam de obedecer à estrutura definida pela especificação API Blueprint.

2.4.4 RAML

RAML ou *RESTful API Modeling Language* é uma linguagem baseada em YAML, e define um *media type "application/raml+yaml"* para a descrição e documentação de APIs REST de maneira que seja facilmente lida e interpretada tanto por humanos como por computadores. Assim, permite que, além de qualquer pessoa conseguir perceber a especificação do serviço, também seja possível de ser interpretada por geradores de código de cliente de APIs, e pelos próprios serviços de modo a criar a documentação de utilizador ou até código de cliente.

Esta especificação introduz também um conceito de tipos de recursos e as suas caraterísticas que permite fazer a caraterização de uma API REST de um modo mais simples, minimizando as repetições necessárias para a sua documentação.

2.4.5 I/O Docs

I/O Docs é uma solução em Node.js para a documentação de serviços *Web* que implementem uma arquitetura REST. Esta especificação permite que para um determinado serviço seja criada uma documentação interativa, com a qual os seus clientes podem interagir de modo a ter conhecimento das diferentes capacidades da API REST em questão.

Através da documentação da API, tanto ao nível dos recursos como dos métodos e dos diferentes parâmetros disponíveis, esta solução irá criar uma interface cliente em JavaScript, onde as diferentes chamadas à API podem ser executadas e onde é possível ter acesso às respetivas respostas.

A definição da API em questão pode ser feita através da escrita de um ficheiro de configuração usando esquemas em JSON, especificando as diferentes propriedades relativas ao serviço, sendo com base nessa mesma configuração que é construída a interface acessível aos clientes.

2.5 *NoBackend*

A componente cliente está a ter cada vez mais importância no desenvolvimento tanto de aplicações móveis como aplicações *Web*, sendo que muitas das tarefas podem já ser aí implementadas sem a necessidade de intervenção de um servidor, no entanto ainda há muitas tarefas que estão dependentes de um servidor, como, por exemplo, autenticação de utilizadores ou até o envio de um simples email [Sla14].

Contrariamente ao que acontece numa abordagem tradicional, onde há um elevado foco na componente de *backend*, uma das abordagens que tem vindo a ser utilizada, denominada de *NoBackend*, defende o desenvolvimento de aplicações focando-se apenas na componente de cliente. A componente de servidor não é completamente ignorada, no entanto, grande parte das tarefas de servidor são abstraídas diretamente no desenvolvimento dos clientes.

O desenvolvimento de serviços através de uma abordagem de *NoBackend*, tem como um dos principais objetivos proporcionar aos programadores uma forma de implementação de *backends* escaláveis, flexíveis e confiáveis para a criação das suas aplicações [Laa13]. Além disso, pretende evitar o esforço de desenvolvimento de uma componente de *backend*, da escolha de tecnologias a usar no mesmo, do tipo de base de dados a usar, dos *endpoints* e métodos a implementar, etc., no qual é dedicado muito do tempo de implementação de uma aplicação *Web*. Todo esse esforço e tempo dedicado ao desenvolvimento de *backend* poderia ser usado para um maior foco de desenvolvimento das componentes que realmente estarão em contacto direto com o utilizador final e que fará toda a diferença na opinião do mesmo sobre a aplicação [Hem13].

Algumas destas soluções oferecem também a possibilidade de trabalhar *offline*, importante para quando os utilizadores pretendem utilizar as aplicações sem acesso à Internet, mas também de modo a reduzir a latência da aplicação. Assim, os utilizadores não têm a necessidade de esperar que os dispositivos comuniquem com os servidores durante a execução normal da aplicação, podendo estes continuar a execução das tarefas pretendidas, sendo os dados guardados localmente e assim que possível sincronizados com o servidor, de um modo transparente e impercetível. Este é um importante ponto na melhoria da experiência de utilização por parte dos utilizadores, já que estes terão a possibilidade de navegar nas aplicações sem os habituais tempos de sincronização de dados que, por vezes, se podem tornar longos e desconfortáveis.

Atualmente existem ferramentas como, por exemplo, Parse, Hoodie e Meteor ou BaasBox, que oferecem aos programadores soluções que lhes permitem fazer o desenvolvimento de aplicações distribuídas de um modo mais eficiente através de abordagens *NoBackend* e *Offline-First* [Sla14].

2.5.1 Parse

O Parse é uma plataforma *Backend as a Service* (BaaS), adquirida pelo Facebook em 2013 [CC13], focada em providenciar serviços de uma forma simplificada para o desenvolvimento de aplicações móveis e *Web*. Este é um serviço inicialmente gratuito, no entanto a partir de um determinado grau de utilização passa a ser pago.

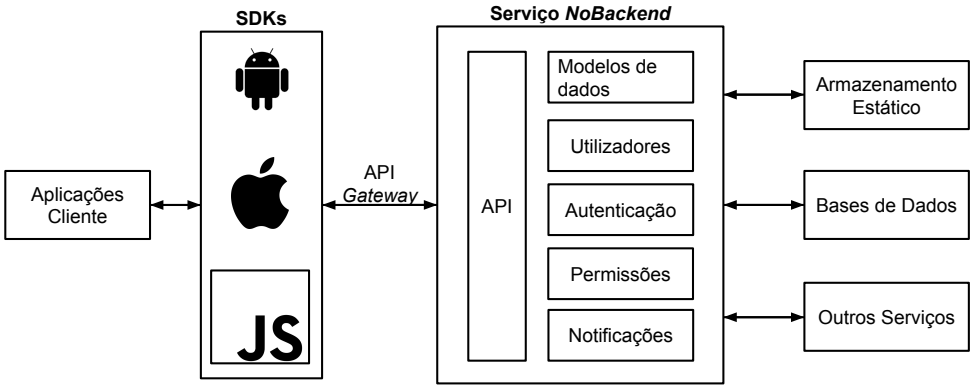

Figura 2.3: Arquitetura de um serviço *NoBackend*

Este serviço oferece aos seus utilizadores essencialmente três tipos de produtos num único pacote: Parse Core, Parse Push e Parse Analytics.

Relativamente ao Parse Core, esta plataforma trata essencialmente do armazenamento, gestão e consulta de dados, das relações entre eles, do controlo de acessos e ainda da gestão de utilizadores. Além disso, esta componente é ainda a responsável pela integração da componente social das aplicações, através de integrações com o Facebook e Twitter, bem como pela possibilidade de integração de código de servidor, agendamento de tarefas e hospedagem.

No que diz respeito ao Parse Push, providencia aos programadores funcionalidades que lhes permite construir e enviar as suas notificações *push* para todos os utilizadores registados ou então para grupos restritos de utilizadores.

Por fim, o Parse Analytics possibilita o controlo dos dados da aplicação, as instalações, os utilizadores ativos, etc.

2.5.2 Hoodie

Hoodie é uma biblioteca *open source*, escrita em JavaScript e implementada com base em CouchDB e Node.js, para o desenvolvimento de aplicações *Web* de uma forma rápida, fácil e acessível, principalmente direcionada para os programadores de *frontend* que pretendem fazer a implementação das suas aplicações com foco em abordagens *Offline First* e *NoBackend*.

Centrando-se numa tecnologia *NoBackend*, tenta facilitar o desenvolvimento de aplicações abstraindo o desenvolvimento de *backend* e proporcionando aos seus utilizadores uma interface *Dreamcode*, possibilitando uma implementação fácil de escrever e ao mesmo tempo de ler e compreender. Além disso, esta solução foca a tendência *Offline First*, privilegiando o armazenamento local dos dados de modo a que os mesmos estejam acessíveis a qualquer momento independentemente da conexão à Internet.

É capaz de oferecer aos seus utilizadores funcionalidades desde a gestão de utilizadores, armazenamento, carregamento, partilha e sincronização de dados, bem como o envio de emails,

apenas através da implementação de *frontend*. Além disso, permite também a extensão das suas funcionalidades através da adição de outros *plugins* existentes.

Desde que seja possível conectar à Internet, permite ainda a implementação com abordagens *real-time*, despoletando eventos aquando das alterações de dados em outras instâncias.

2.5.3 Meteor

Meteor é uma plataforma de desenvolvimento de aplicações *Web* e móveis em JavaScript puro, integrando todos os componentes necessários para a implementação das aplicações, bibliotecas, bases de dados e *frameworks* de *frontend* e *backend*.

Além do acesso quase direto à base de dados, esta solução tem um grande foco no desenvolvimento de aplicações com uma abordagem *real-time* permitindo a transmissão das alterações de dados em tempo real entre os diferentes clientes. Além disso, e de modo a diminuir o efeito da latência da rede, o Meteor armazena também os dados localmente de modo a oferecer uma melhor experiência de utilização, para que a alteração de dados locais, provoque a atualização das vistas automaticamente e em tempo real, não havendo o tempo de espera da atualização dos dados na base de dados no servidor.

O Meteor permite ainda a gestão de dados através de coleções de MongoDB, gestão de utilizadores, dados de sessão, *login* com contas de Facebook, Google, GitHub e Twitter. Além disso, o Meteor, possui uma *framework* de *frontend* que permite a criação de *templates* capazes de interagir com dados dinâmicos e com os eventos recebidos, de modo a fazer a gestão dos dados renderizados.

2.5.4 BaasBox

BaasBox ou *Backend as a Service in a Box*, tal como o nome indica é um serviço disponibilizado como *Backend as a Service*, *open source*, que tem como principal foco a implementação de aplicações *Web* de uma forma rápida, focando o desenvolvimento nas componentes de *frontend*, evitando as dificuldades e o esforço de desenvolvimento de *backend*.

Este serviço oferece um conjunto de funcionalidades, todas incluídas num mesmo conjunto, semelhante a uma caixa, sendo apenas necessário ter instalado uma JVM de modo a que o BaasBox possa estar operacional.

Implementado em Java e também com algum código em Scala, este serviço utiliza por base a Play Framework, *framework* para a implementação de serviços REST, por forma de construção deste tipo de serviços, e incorpora uma base de dados OrientDB, por forma a guardar todos os recursos sob a forma de documentos JSON.

Tal como outras ferramentas deste contexto, possui SDKs para a utilização da ferramenta quer através do desenvolvimento em JavaScript, como também em Android e iOS, por forma a utilizar todas as funcionalidades disponibilizadas pelo servidor, acessíveis aos clientes.

Relativamente ao serviço, este oferece uma área de administração através de uma interface *Web*, através da qual é possível fazer a gestão de todo o serviço, desde modelos de dados, gestão

de utilizadores e respetivos papéis dentro da aplicação. Além disso, este serviço permite também fazer autenticação através de serviços sociais externos tais como o Facebook e o Google+, bem como o envio de notificações para dispositivos móveis Android e iOS.

2.5.5 Comparação

Através da tabela 2.2 e com a descrição das soluções em questão, é possível perceber que as *frameworks* Parse e BaasBox são as soluções mais completas para o desenvolvimento deste tipo de serviços, possuindo um maior suporte a nível de plataformas de desenvolvimento, bem como uma maior maturidade da solução oferecida. Apesar de não possuírem suporte para desenvolvimento através de abordagens *real-time*, estas soluções são as que oferecem funcionalidades mais maduras tanto relativamente à gestão e manipulação de coleções de dados, bem como gestão de utilizadores, acessos, e outras funcionalidades tais como capacidade de manipulação de dados geo-espaciais ou até gestão de ficheiros estáticos. Ao nível da qualidade da documentação, também estas apresentam uma documentação bem detalhada, intuitiva, ilustrada através de exemplos, e bastante acessível aos utilizadores.

Relativamente ao Hoodie, atualmente é uma solução que se encontra ainda em desenvolvimento, e que apesar de apresentar potencial para uma futura boa escolha de utilização, neste momento ainda possui pouco suporte ao nível de documentação, dificultando o desenvolvimento de uma solução relativamente complexa.

Por fim, a Meteor, é também uma solução bastante completa, proporcionando mapeamento direto dos dados para os *templates*, com capacidade de atualização automática dos mesmos. Esta ferramenta integra todos os componentes necessários para a implementação de aplicações *Web*, desde a gestão de bases de dados até a componentes de manipulação de *frontend*.

Tabela 2.2: Comparação de *frameworks* de *NoBackend*

	SDKs	**Preço**	***Real-time***	***Offline First***	**Dados**	**Notificações Push**	**Autenticação Terceiros**
Parse	Android, iOS/OS X, .Net, JavaScript, PHP, Unity	Gratuito e Pago	Não	Não[6]	JSON Documents	Sim	Facebook, Twitter
Meteor	JavaScript	Gratuito	Sim	Não[7]	MongoDB	Não	Facebook, Google, GitHub, Twitter
Hoodie	JavaScript, iOS	Gratuito	Sim	Sim	CouchDB	Sim	Não
BaasBox	iOS, Android, JavaScript	Gratuito	Não	Não	OrientDB	Sim	Facebook, Google

2.6 Micro Serviços

> "Micro serviços é um padrão de arquitetura de software que implica a distribuição de sistemas como um conjunto de serviços colaborativos, muito pequenos, granulares e independentes." [Hof14]

Este padrão propõe uma evolução no desenvolvimento de aplicações *Web*, passando a implementação de serviços complexos, de uma arquitetura monolítica, na qual toda a aplicação é desenvolvida como um sistema único, englobando todos os componentes, para uma arquitetura constituída por um conjunto de micro serviços. A principal ideia por detrás deste padrão de arquitetura é a conceção de sistemas grandes, complexos e duradouros como um conjunto de serviços coesos que evoluem ao longo do tempo [Ric14].

Com isto, um micro serviço pode ser interpretado como um serviço leve, independente, de responsabilidades relativamente granulares que oferecem funções únicas e são ainda capazes de colaborar com outros serviços, necessitando para isso, de possuir uma interface muito bem definida [NSs14].

Deste modo, é possível dotar um sistema de um conjunto de serviços independentes do sistema, tanto ao nível da linguagem de programação, como ao nível do ambiente de produção, modelos de dados, etc., sendo capazes de executar em processos independentes. Assim, para um determinado sistema, é possível adaptar os seus requisitos ao tipo de implementação, tecnologias ou modelos de dados mais adequados ao cumprimento dos objetivos, independentemente do serviço base do sistema, desde que os micro serviços desenvolvidos obedeçam às regras estabelecidas para a interface de ligação entre os diferentes componentes.

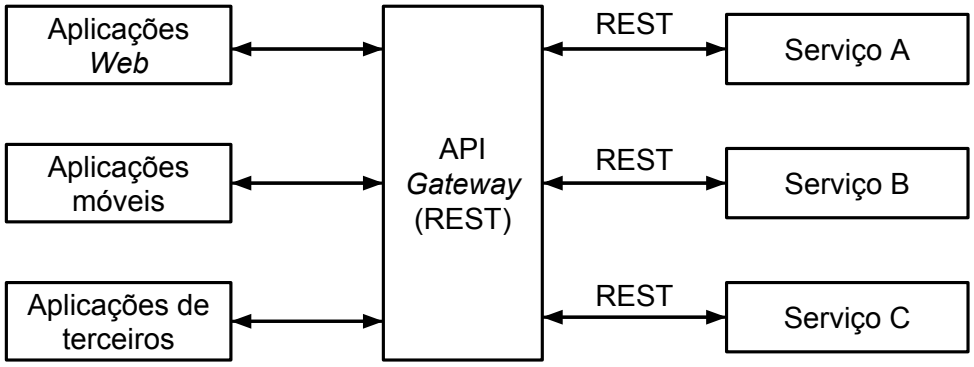

Figura 2.4: Possível arquitetura de um sistema baseado em micro serviços

[6]Não é possível fazer a gestão dos dados em modo *offline*, no entanto em alguns SDKs é possível fazer o armazenamento de dados em modo *offline*, sendo estes sincronizados assim que possível.

[7]Os dados são guardados e atualizados localmente de modo a evitar a perceção de latência da rede, mas não existe mecanismo funcionamento *offline* garantindo uma posterior sincronização de dados

2.6.1 Motivação

O desenvolvimento de aplicações *Web* obedecendo a uma arquitetura monolítica possui vantagens nomeadamente na complexidade de desenvolvimento, dado que grande parte das ferramentas são orientadas ao desenvolvimento de uma única aplicação, tornam-se mais fáceis de testar, de distribuir [Ric14] e até de escalar, através da execução de múltiplas instâncias por detrás de um sistema de balanceamento de carga [NSs14]. Apesar das vantagens da utilização de uma arquitetura monolítica, a motivação para o uso de micro serviços centra-se essencialmente nas desvantagens da implementação de um serviço *Web* através de uma arquitetura monolítica, na qual todo o sistema é construído como um só.

Em primeiro lugar, o desenvolvimento através de arquitetura monolítica, pode tornar-se mais dificultado quando a aplicação se torna complexa e significativamente grande. Nestas situações, a compreensão, manutenção e até a distribuição do sistema torna-se mais difícil, sendo que para cada implantação das alterações é necessário compilar e executar todo o sistema, o que se pode tornar uma tarefa complexa, com riscos, demorada e que requer a coordenação de várias entidades [Ric14].

Devido à integração de todas as funcionalidades no mesmo código fonte este torna-se consideravelmente grande, o que é propício ao declínio da sua qualidade, provocando assim uma redução da produtividade da equipa, que necessita de desenvolver de modo dependente, aumentando assim o esforço necessário por cada um dos membros para a implementação das suas tarefas [NSs14].

Além disso, uma arquitetura monolítica pode apenas ser escalada em uma dimensão. É possível aumentar o volume de transações aumentando o número de cópias em execução da mesma aplicação, no entanto não é possível escalar com o aumento do volume de dados, dado que cada uma das cópias da aplicação terá acesso ao mesmo conjunto de dados [NSs14].

Por fim, com uma arquitetura monolítica, a adoção de novas ferramentas e tecnologias está muito restrita, porque por um lado o sistema está limitado ao ambiente inicial de desenvolvimento e à linguagem utilizada, e por outro lado, a migração um determinado serviço para outro tipo de tecnologia obriga a um grande esforço de implementação.

São estes problemas de uma arquitetura monolítica, que o padrão de arquitetura de micro serviços tenta resolver.

2.6.2 Vantagens

Em primeiro lugar, uma das principais vantagens deste padrão de arquitetura, que está também na origem de outras vantagens, é o tamanho de cada um dos serviços. Tal como o nome do padrão indica, cada um dos micro serviços é relativamente pequeno, tornando-se mais facilmente compreensíveis para os programadores, propiciando assim uma maior produtividade, tanto ao nível do programador, como ao nível da capacidade de processamento dos IDEs utilizados, já que estarão a lidar com aplicações mais leves [Ric14].

Um outro ponto a ter em conta é o facto de cada um dos micro serviços ser independente entre si, assim cada um dos serviços pode ser implantado individualmente, sem as normais dependências existentes, abrindo a possibilidade de adoção de novas tecnologias, ferramentas e metodologias, tendo em conta os requisitos a serem cumpridos. Este fator propicia e facilita também as equipas de desenvolvimento tendo em conta os diferentes micro serviços, possibilitando que cada um dos componentes do sistema seja implementado em modo independente, desde que as normas estabelecidas para a interface entre os diferentes módulos seja cumprida.

Este tipo de arquiteturas proporciona também um maior grau de escalabilidade do sistema, pois cada um dos serviços, independentes entre si, pode ser individualmente escalado, estando cada um dos serviços relacionado a apenas uma fração do conjunto de dados, diminuindo as suas necessidades em termos de memória.

Além das falhas já enunciadas, esta arquitetura permite também uma maior tolerância a falhas, já que quando ocorre uma falha num dos micro serviços, o sistema "central", bem como os outros serviços continuam em execução, não levando à falha do sistema como um todo, ao contrário de uma arquitetura monolítica em que uma falha leva a que todo o sistema fique indisponível. Além disso, é possível testar mais facilmente a funcionalidade ao que cada um dos micro serviços se refere mais facilmente, devido à baixa complexidade dos mesmos.

2.6.3 Desvantagens

Apesar das vantagens da implementação de uma arquitetura de micro serviços, existem algumas desvantagens na sua utilização, que devem ser mantidas em conta na decisão de implementação de um sistema destes.

Em primeiro lugar, a complexidade de criação de um sistema distribuído, o estabelecimento de normas e critérios de comunicação, a quantidade de comunicação necessária entre os processos, e a complexidade de execução do sistema como um todo, pode tornar o arranque da implementação de um projeto deste tipo bastante difícil e requerer um grau de esforço considerável. Além disso, ao nível das ferramentas, neste momento os IDEs não suportam explicitamente o desenvolvimento de aplicações de um modo distribuído tal como acontece numa arquitetura de micro serviços [Ric14].

Relativamente à fase de testes do sistema, esta torna-se também muito mais complicada, devido à existência de vários processos independentes. Apesar do sistema funcionar igualmente como um único serviço e as funcionalidades poderem ser testadas tal como acontece com um sistema monolítico, devido à existência de múltiplos micro serviços independentes, torna-se também necessário implementar testes de integração, de modo a testar a ligação entre os diferentes serviços. Por outro lado, enquanto num sistema monolítico todos os erros podem ser apanhados numa única aplicação, num sistema de micro serviços é mais difícil fazer o *debug* de erros, e a descoberta de eventuais falhas no sistema [Hof14].

O desenvolvimento tendo em conta esta arquitetura implica que sejam ainda criados acordos entre as diferentes equipas de desenvolvimento relativamente às interfaces de comunicação entre os diferentes micro serviços. Apesar de independentes, os micro serviços devem manter uma

interface explícita para a ligação entre eles, sendo comunicadas quaisquer alterações que ocorram por forma a que a comunicação e a sincronização entre os mesmos seja assegurada [Avr14].

A utilização deste padrão de arquitetura possui ainda alguns desafios. Um deles está relacionado com a decisão do ponto de rotura entre uma arquitetura monolítica para uma arquitetura de micro serviços, dado que num ponto de desenvolvimento inicial de um determinado projeto pode não ser favorável utilizar esta abordagem, no entanto, poderá haver um ponto em que essa migração seja favorável [Ric14]. Além disso, é necessário também decidir qual a granularidade dos micro serviços relativamente à carga de trabalhos, funcionalidades pelas quais estarão responsáveis bem como o conjunto de dados com os quais terão de lidar, para que haja um balanceamento de carga correto entre os diferentes micro serviços e uma divisão lógica ao nível das funcionalidades.

2.7 Resumo

Em primeiro lugar, através da análise dos conceitos relativos ao estilo de arquitetura REST e das *frameworks* REST já existentes, foi possível perceber que, apesar de já existirem boas soluções para o desenvolvimento deste tipo de serviços, nem todas permitem seguir os princípios e boas práticas definidas pela arquitetura.

Esta arquitetura apresenta como um dos principais princípios, a utilização de interface uniforme na disponibilização dos serviços, no entanto muitas das soluções existentes apesar de passarem para o exterior uma representação baseada em recursos, não levam esse conceito a um nível de servidor, permitindo que o programador defina as diferentes rotas, deixando a implementação da lógica de cada uma delas e de cada um dos métodos ao encargo dos programadores, aumentando assim a flexibilidade mas ao mesmo tempo facilitando a introdução de práticas menos aconselhadas no desenvolvimento deste tipo de serviços, abrindo margem para o incumprimento de princípios tidos na base desta arquitetura.

Relativamente à restrição HATEOAS, poucas soluções permitem de um modo simples, fazer com que a API em desenvolvimento obedeça a este princípio de uma forma automática ou com um reduzido esforço necessário de implementação. Assim, é raro encontrar soluções que permitam construir uma API totalmente navegável, permitindo ter ligação entre os diferentes recursos associados.

Uma das boas práticas também definidas pela arquitetura REST é o desenvolvimento de documentação do serviço, por forma a facilitar o acesso e a compreensão do serviço pelos clientes. Existem já alguns métodos e ferramentas que permitem produzir uma documentação automática das APIs, no entanto, grande parte das soluções analisadas não possuem suporte para este tipo de funcionalidades, à exceção de Django REST Framework e Loopback, sendo este um dos principais problemas encontrados nas soluções analisadas.

No que diz respeito às soluções para a implementação de serviços com a abordagem *NoBackend*, existem já boas soluções, nomeadamente Parse ou BaasBox, no entanto apenas a BaasBox é totalmente gratuita, sendo a Parse gratuita mas a partir de um determinado grau de

utilização é paga. Além disso, as soluções *NoBackend* existentes, tais como as *frameworks* REST, possuem ainda falhas, no que diz respeito ao cumprimento das normas e boas práticas de uma arquitetura REST. Também este tipo de soluções é deficiente no que se refere à documentação da API, não possuindo foco em práticas de documentação automática dos serviços, por forma a proporcionar aos seus clientes uma forma mais facilitada de compreensão das APIs produzidas.

Uma tendência recente no desenvolvimento de serviços *Web* é a utilização de uma arquitetura de micro serviços, por forma a aumentar a escalabilidade e a modularidade do sistema. Apesar desta tendência, ainda não existem *frameworks* REST ou *NoBackend* direcionadas a este tipo de arquitetura, ou que, pelo menos, incitem a utilização de boas práticas que permitam a fácil evolução de um serviço para um conjunto de micro serviços, permitindo a fácil separação das funcionalidades ou recursos disponíveis.

Relativamente à linguagem de programação a utilizar, o JavaScript, foi possível perceber que grande parte das soluções analisadas ainda não fazem uso dos conceitos e funcionalidades mais recentes apresentadas por esta linguagem e que têm recentemente revolucionado o estilo de programação de JavaScript, como, por exemplo, o uso de *promises*, *generators*, iteradores, funções assíncronas, *decorators*, classes, etc.

Capítulo 3

Problema e Solução

Após o estudo do estado da arte relativo às diferentes temáticas envolventes aos serviços REST e a análise das diferentes limitações existentes nas soluções já existentes, esta dissertação apresenta uma proposta para a implementação de uma *framework* para o desenvolvimento de serviços REST, que seja capaz de ter em conta os diferentes problemas encontrados nas soluções existentes fazendo dos mesmos fatores de diferenciação.

3.1 Descrição do Problema

De um modo geral, esta dissertação teve como principal problema de estudo, o desenvolvimento de uma *framework open source* na linguagem JavaScript, mais propriamente através da tecnologia Node.js que permita fazer a implementação de serviços REST.

Atendendo ao principal problema enunciado podem ser apresentados mais em pormenor outros problemas associados que também serão alvo desta dissertação:

- Aplicabilidade dos *standards* e restrições definidas para a arquitetura REST.
- Construção de uma *framework* que permita construir serviços obedecendo às restrições do princípio de interface uniforme.
- Disponibilização de documentação do serviço bem como uma interface HATEOAS.
- Utilização de mecanismos de *cache*[1] com foco na performance do serviço
- Capacidade de disponibilização de um serviço genérico capaz de ser integrado com aplicações *NoBackend*.
- Transmissão de alterações dos dados para os clientes do serviço em tempo real.
- Integração das novas caraterísticas tecnológicas da linguagem utilizada.

[1] Componente que armazena dados para que os futuros pedidos a esses dados possam ser respondidos mais rapidamente

3.1.1 Arquitetura REST

No desenvolvimento desta *framework* foi tido em conta o estilo de arquitetura REST, no entanto, dado que este é um estilo de certa forma flexível e apenas baseado num conjunto de princípios e boas práticas apresentados por Roy Fielding há cerca de 15 anos, a aplicabilidade de cada um desses aspetos na atualidade, atendendo à evolução das tendências no desenvolvimento deste tipo de serviços, foi um dos principais problemas atendidos.

3.1.2 Interface Uniforme

Um dos princípios apresentados por esta arquitetura, denominado interface uniforme, salienta que as APIs devem ser baseadas em recursos, componentes da aplicação possíveis de ser comunicadas aos clientes e identificáveis a partir de um URL único. Apesar de os serviços passarem para o seu exterior a ideia de que cada um dos recursos funciona como uma entidade isolada, muitas das vezes na implementação do serviço nem sempre essa divisão e modularidade é facilmente compreensível e bem delimitada. Este é um dos problemas que mereceu uma especial atenção, tentando levar o conceito de recurso ao modo mais modular possível, de modo que não só para os clientes esse recurso seja encarado como uma entidade. Esta dissertação pretende que para toda a extensão do serviço os recursos sejam claramente identificáveis como uma estrutura modular que possa ser utilizada também internamente como uma entidade orientada a objetos.

3.1.3 Documentação do serviço

Um dos problemas recorrentes deste tipo de serviços é a ausência de documentação, o que é fomentado pela inexistência de normas por parte da própria arquitetura, tornando assim difícil a compreensão do serviço pelas entidades externas à sua implementação. Apesar da existência de algumas abordagens para a produção de documentação, um dos problemas que prevalece é o esforço adicional necessário para a construção da mesma e, dado não ser obrigatória para o correto funcionamento do serviço, acaba muitas vezes por não ser disponibilizada. Neste sentido, um dos problemas e objetivos a abordar por esta implementação é o desenho de uma solução através da qual a disponibilização deste tipo de documentação possa ser feita de um modo automático, sem necessidade de intervenção dos programadores do serviço.

3.1.4 *HATEOAS*

Outro dos problemas abordados foi a capacidade de uma API ter a capacidade de ser automaticamente descoberta e explorada, atendendo à restrição *Hypermedia as the Engine of Application State*, permitindo que a partir de um determinado recurso seja possível descobrir a localização de todos os recursos que lhe estão associados. Esta característica pode permitir que os clientes da API, em qualquer ponto, possam eles próprios compreender a forma como podem interagir como o mesmo, navegar entre os diferentes recursos e entre as diferentes

funcionalidades que cada um deles disponibiliza. Apesar de realmente útil, este princípio é ainda um ponto em aberto no desenho de serviços REST, não existindo um acordo quanto à forma ideal para a sua concretização.

3.1.5 Performance

Uma das preocupações existentes com os serviços REST é a performance do sistema especialmente aquando da receção de um elevado número de pedidos ao serviço. Muitos dos pedidos que são feitos ao serviço, muitas das vezes, foram já feitos pelo cliente ou até por outros clientes, e assim, a carga de execução poderia ser evitada. É neste aspeto que a solução desenhada tentou intervir, incorporando mecanismos de *cache* a vários níveis no fluxo de tratamento dos pedidos.

3.1.6 *NoBackend*

Atualmente, uma das tendências que começa a ser alvo de interesse pela comunidade de desenvolvimento de aplicações móveis e *web* é a construção de aplicações através de uma abordagem *NoBackend*[2]. Apesar da existência desta tendência existe ainda dificuldade em integrar essa abordagem com uma abordagem tradicional na qual todos os recursos são construídos pelo serviço. Assim, este é um dos problemas a ser focado nesta dissertação, de modo a possibilitar a construção de serviços tradicionais mas também a disponibilização de uma interface genérica capaz de ser utilizada através desta abordagem mais orientada ao cliente, na qual, apesar de existir uma componente *backend* no sistema, o programador não precisa de se preocupar com a sua implementação.

3.1.7 Comunicação em tempo real

Ainda com foco na performance do sistema e também no acompanhamento das atuais tendências no desenvolvimento de aplicações móveis e *web*, a solução implementada pretende possibilitar a construção de APIs REST nas quais além da abordagem tradicional para o acesso aos dados, permite também que as alterações nos dados sejam transmitidas em tempo real, sem que seja necessário a aplicação cliente estar constantemente a sobrecarregar o serviço com novos pedidos que em grande parte dos casos serão inúteis pela inexistência de alterações nos dados.

3.1.8 Tecnologias

Por fim, atendendo à linguagem de programação utilizada para o desenvolvimento da framework e aos seus constantes desenvolvimentos de especificação, é pretendido que seja possível aos programadores dos serviços sejam capazes de usar as tendências mais atuais, que apesar de poderem introduzir um esforço de aprendizagem inicial, permitem fazer a construção

[2] Abordagem que permite abstrair a tarefas de *backend* no código de cliente

dos seus serviços de uma forma mais simplificada, com utilização de código de uma forma mais limpa e ao mesmo tempo fazer uso das mais recentes tendências tecnológicas.

3.2 Solução

Por forma a facilitar a compreensão da *framework*, foi desenhada uma solução que pretende que o número de conceitos que o programador necessita de interiorizar seja o mínimo possível e, ao mesmo tempo, permita uma separação organizada das diferentes responsabilidades do sistema. Assim, de um modo geral, a *framework*, assenta em quatro conceitos:

Input

É a entidade através da qual é construído um esquema de validação de dados. Posteriormente este esquema pode ser associado tanto a *Models* como a *Resources* servindo como forma de validação dos dados recebidos em cada um dos métodos.

Model

É a única entidade que tem a responsabilidade de fazer a conexão da *framework* com a base de dados, tendo assim, obrigatoriamente, de ser implementado de acordo com o tipo de base de dados alvo. Pode ser considerado uma espécie de ORM, no qual objetos da base de dados estão representados através de objetos de JavaScript que ao mesmo tempo possuem métodos para a manipulação dos seus estados internos.

Resource

Para a *framework*, uma *Resource* corresponde ao que numa arquitetura REST é também considerado um recurso, representando uma entidade em que sobre a mesma podem atuar um conjunto de métodos correspondentes aos diferentes tipos de pedidos HTTP que podem ser efetuados. Deste modo, esta é a entidade, em muitas outras soluções considerada o controlador e, que faz a ligação entre o *Router* do serviço e os respetivos *Models* quando necessária a interação com a base de dados.

Aliado a esta entidade existe ainda uma outra, *GenericResource*, que basicamente é uma subclasse na qual os principais métodos já se encontram implementados, de acordo com o objetivo base de cada um dos métodos, atendendo a uma arquitetura REST, possibilitando assim que, seja apenas necessário associar um *Model* para que se torne funcional.

Router

É a entidade de mais baixo nível no serviço responsável pela receção dos pedidos HTTP e o encaminhamento para a respetiva *Resource* e método correspondente. Oferece a possibilidade de lhe serem associadas *Resources*, para que, deste modo, sejam disponibilizadas pela API.

Também a esta entidade está associada uma outra de contexto genérico, o *GenericRouter* que, para além de oferecer todas as funcionalidades de um *Router*, permite que o serviço funcione através de uma perspetiva de *NoBackend*, no qual é possível que as *Resources* sejam criadas em tempo de execução de acordo com o tipo de pedidos recebidos pelo serviço.

3.2.1 Narrativas de Utilização

Como ponto de partida para o desenho de uma solução foi criada uma lista de funcionalidades às quais o programador deverá ter acesso de modo a poder construir o seu próprio serviço.

As narrativas de utilização (*user stories*) a seguir apresentadas foram organizadas de acordo com as principais entidades envolvidas na *framework*.

Input

- Como programador quero definir um esquema de validação de dados.
- Como programador quero associar atributos simples (*string*, *int*, *boolean*, *date*) ao Input.
- Como programador quero associar diferentes tipos de propriedades de validação de dados a cada um dos atributos definidos pelo *Input*.
- Como programador quero definir relações com outros modelos de dados.

Model

- Como programador quero definir uma classe de acesso à base de dados.
- Como programador quero associar um esquema de validação dos dados ao *Model* definido.
- Como programador quero definir qual é o nome com que o *Model* está representado na base de dados com a qual interage.
- Como programador quero definir atributos de pesquisa predefinidos para o acesso a conjunto de dados do *Model* em questão.
- Como programador quero definir a forma como os dados são representados nos dados devolvidos pelo *Model*.
- Como programador quero associar ações personalizadas ao *Model*.

Resource

- Como programador quero definir uma classe *Resource*.
- Como programador quero associar um esquema de validação dos dados à *Resource* definida.
- Como programador quero associar uma classe *Model* à *Resource* a ser definida.
- Como programador quero definir quais as operações que são possíveis de ser utilizadas pelos clientes.
- Como programador quero definir qual o método de autenticação associado à *Resource*.

- Como programador quero definir quais os grupos de utilizadores que podem aceder à *Resource*.
- Como programador quero definir atributos de pesquisa predefinidos para o acesso a conjunto de dados da *Resource* em questão.
- Como programador quero definir a forma como os dados são representados nos dados devolvidos pelo *Resource*.
- Como programador quero definir a forma de renderização dos dados devolvidos pela *Resource*.
- Como programador quero associar ações personalizadas à *Resource*.
- Como programador quero definir um identificador para a *Resource*, que poderá ser usado como caminho de acesso para os clientes.
- Como programador quero documentar as *Resources* definidas.

Além das já mencionadas, existe ainda um conjunto de narrativas de utilização mais gerais, que não são especificamente associadas a um único conceito dos já referidos, mas sim associado ao serviço na sua generalidade.

- Como programador quero disponibilizar a documentação das *Resources* através de uma interface HATEOAS.
- Como programador quero disponibilizar a documentação das *Resources* através de uma interface em HTML.
- Como programador quero captar as alterações dos dados e transmiti-las para os clientes do serviço em tempo real.
- Como programador quero permitir a criação de recursos no serviço em tempo de execução.

3.2.2 *Dreamcode*

Após a definição das narrativas de utilização correspondentes às principais funcionalidades da *framework* a ser desenvolvida, e antes mesmo da definição da organização da mesma, esta dissertação passou por uma etapa de planeamento relativamente à interface programática, tendo sido definido o *dreamcode*, ou código ideal, para a *framework*, representando a forma como os programadores poderiam interagir de modo a utilizar as diferentes funcionalidades da mesma.

Esta fase pode ser considerada uma das fases mais importantes desta dissertação. Além deste processo fazer uma ligação entre as narrativas de utilização, tal como elas foram apresentadas na secção anterior, para uma sintaxe programática, na qual é possível ao programador perceber a forma como as mesmas poderão ser aplicadas em código, este processo permitiu também incutir

no desenvolvimento uma preocupação constante na forma como as funcionalidades são implementadas e ao mesmo tempo disponibilizadas para os seus utilizadores, neste caso, os programadores.

Input Através da análise de algumas soluções já existentes para a definição de esquemas de dados, como é o caso da módulo *Waterline*[3] para Node.js, foi estruturada uma sintaxe para a definição dos esquemas de dados na solução a desenvolver. Assim, o esquema de validação de dados recebidos pela API pode ser definido através de um objeto no qual são incluídos os diferentes atributos e as diferentes propriedades relativas a cada um dos atributos.

```
var restaurantInput = new Input({
    name:     {type:"string", required:true, regex : /restaurante_*/i},
    address:  {type:"string"},
    capacity: {type:"number", valid:capacityValidator},
    expires:  {type:"date"},
    products: {type:"collection", model:'Product'}
});

var capacityValidator = function(val){
  if(val < 0)
      throw "Must be higher than 0";
  else
      return true;
};

restaurantInput.valid(data);
```

Listing 3.1: Definição de um esquema de validação de dados

Resource Relativamente à entidade *Resource*, o código idealizado contempla a implementação de uma classe *Resource*, em que os seus métodos, estáticos e de instância, correspondem aos diferentes métodos HTTP que podem ser recebidos pelo servidor para um determinado recurso. Desta forma, é possível que a *Resource* implementada possa ser usada tanto em conjunto com um *router*, como imperativamente em qualquer ponto da aplicação.

```
@Input(...)
@Query(...)
@Output(...)
@Name(...)
@Documentation({
    title: 'title of the resource'
    description: 'description of the resource'
```

[3] https://github.com/balderdashy/waterline

```
})
@Model(ModelClass)
@Roles(['ALL'])
@Authentication('basic')
@Methods(['patch', 'put'])
@Format(JSONFormat)
class RestauranteResource extends GenericResource{

let restaurantArray = RestaurantResource.fetch(requestData);

let restaurantObject = new RestaurantResource(requestData);
let restaurantObject = RestaurantResource.fetchOne(requestData);

restaurantObject.put(requestData);
restaurantObject.patch(requestData);
restaurantObject.delete(requestData);

restaurantArray.render();
restaurantArray.render(HTMLRenderer);

RestaurantResource.valid(data);

let produtos = restaurante.post_cleanBox(requestData);
let produtos = restaurante.get_products(requestData);
```

Listing 3.2: Utilização de uma *Resource*

Models Tal como na *Resource*, também os *Models* seguem uma estratégia orientada a objetos, possuindo uma interface de utilização semelhante, proporcionando assim uma interface uniformizada para estas duas entidades o que permite também reduzir o custo de aprendizagem de utilização para o programador. Assim, os métodos estáticos e de instância, bem como as ações personalizadas possuem a mesma forma de utilização, diferenciando-se das *Resources*, por exemplo, na forma como é possível fazer a atualização dos dados.

```
@Input(restauranteInput)
@Name('restaurant')
@Query({
        _sort: ['-name','-address'],
        _page_size: 15,
        _filter: { street: "That Street"}
    })
@Output({
        _fields: ['name','adress'],
        _embedded: ...
    })
class RestaurantModel extends GenericModel{}
```

```
13
14 let restaurantArray = RestaurantModel.fetch(requestData);
15 let restaurantObject = RestaurantModel.fetchOne(requestData);
16
17 let restaurantObject = new RestaurantModel(requestData);
18
19 restaurantObject.put(requestData);
20 restaurantObject.patch(requestData);
21 restaurantObject.delete(requestData);
22
23 restaurantObject.obj.name= "newRestaurantName";
24 restaurantObject.oldObj.name != restaurantObject.obj.name;
25 restaurantObject.save();
26
27 RestaurantModel._input.valid(restaurante.obj);
28 restaurantObject.valid();
29 restaurantObject.valid(restaurantInput);
```

Listing 3.3: Utilização de um *Model*

Router Relativamente ao *Router*, este tem o objetivo de servir de entidade de registo das diferentes *Resources* que serão disponibilizadas pelo serviço, bem como do caminho através do qual estas serão disponibilizadas.

```
1  let router = new Router(
2          [
3              { resource:RestaurantResource, path:"restaurant"},
4              { resource:ProductResource, path:"product"}
5          ]
6      );
7  let router = new Router({ resource:ProductResource, path:"product"});
8
9  router.register(
10         [
11             { resource:RestaurantResource, path:"restaurant"},
12             { resource:ProductResource, path:"product"}
13         ]
14     );
15 router.register({ resource:ProductResource, path:"product"});
```

Listing 3.4: Utilização de um *Router*

Tipos de Pedidos

Embora esta secção não seja relacionada com o interface programática com que o programador terá de interagir, está de igual forma relacionada com a forma como os seus clientes poderão

interagir com o serviço.

Tal como o programador pode indicar propriedades de pesquisa (*query*) e de representação das respostas (*output*) diretamente nos *Models* e nos *Resources*, funcionando estes de uma forma predefinida para todos os pedidos a estas entidades, também no momento da requisição do serviço através de pedidos HTTP os clientes poderão enviar esse tipo de propriedades.

```
1  http://api.path/?_filter={"name":"Filipe"}
2
3  http://api.path/?_sort=["-name","age"]
4
5  http://api.path/?_fields=["id", "name", "age", "BI"]
6
7  http://api.path/?_embedded=["photos", "posts"]
8
9  http://api.path/?_page=1&_page_size=15
10
11 http://api.path/?_format=json
```

Listing 3.5: Tipo de propriedades possíveis de serem enviadas pelos pedidos

Apenas com a definição do código ideal da *framework*, foi possível, embora de um modo geral, ter uma visão da forma como o programador irá aceder às diferentes funcionalidades e a partir daí evoluir a conceção da *framework* sempre com foco em proporcionar uma sintaxe simples e intuitiva bem como fomentar boas práticas de programação para os seus utilizadores.

3.2.3 Arquitetura

Nesta secção é apresentada a arquitetura da solução desenvolvida por esta dissertação. Após já ter sido explicada em pormenor cada uma das entidades da *framework* é também apresentada a forma com as mesmas se interligam, numa visão geral da arquitetura, bem como a forma como algumas das funcionalidades são encaixadas no sistema desenvolvido.

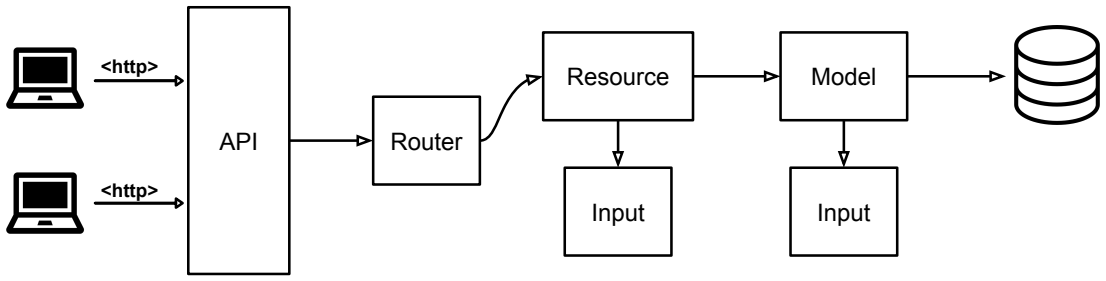

Figura 3.1: Visão geral da arquitetura da *framework*

Mapeamento Objeto-Relacional e Mapeamento Objeto-Recurso

Um aspeto importante no desenho desta implementação é a existência de um mapeamento direto entre as diferentes componentes existentes.

Tal como é comum haver um mapeamento objeto-relacional para a representação das informações existentes na base de dados através de uma perspetiva orientada a objetos adaptada à linguagem de programação utilizada, também esta *framework*, através da entidade *Model* pretende fazer esse mesmo mapeamento, tornando assim mais fácil e intuitiva a manipulação e a compreensão dos dados.

Além deste tipo de mapeamento, a *framework* assenta ainda num mapeamento objeto-recurso, no qual, os objetos que representam as *Resources* possuem métodos correspondentes a cada um dos tipos de métodos que podem ser tratados pela API, possibilitando assim uma abstração maior entre a lógica do serviço e a camada HTTP. Além disso, permite também uma correspondência praticamente direta desde os métodos HTTP, aos métodos das *Resources* e até aos métodos dos *Models*.

Assim, através deste desenho de solução, poderemos considerar a entidade *Model* como um ORM para a base de dados e a entidade *Resource* como um *Object Resource-Mapping* para os pedidos recebidos pela API.

Modularidade e Agnosticidade

Esta solução pretende promover o desenvolvimento de serviços de uma forma simples e obedecendo a boas práticas de programação e, atendendo a esse objetivo, tenta promover a separação das diferentes responsabilidades pelas diferentes entidades, como foi já mencionado anteriormente. A solução tenta promover este fator a um nível superior, nos quais, para além da separação de responsabilidades, mantêm o seu comportamento independentemente do tipo de implementação usado ao nível do acesso da base de dados ou ainda do tipo de *middleware* usado para o encaminhamento dos pedidos.

Ao nível do acesso à base de dados é pretendido que a *framework* seja independente do tipo de base de dados, e deste modo todas as responsabilidades de acesso à mesma são encarregadas a esta entidade. Através desta separação é possível que a *framework* consiga interagir com diversos tipos de bases de dados, de modo transparente, obrigando apenas a que a entidade *Model* seja estendida de acordo com a base de dados com a qual irá interagir e ao mesmo tempo respeitando a sintaxe pretendida para este tipo de entidade.

No que diz respeito ao *Router*, a *framework* pretende também ser agnóstica de *middleware*, ou seja, é possível que esta camada seja implementada de acordo com o tipo de *middleware* que o programador pretender utilizar, havendo apenas a necessidade de adaptar a componente do *Router* que tem ligação direta com a componente de *middleware*. Isto é possível apenas porque a *Resource* é independente da sua camada mais superior representada pelo *Router*. Embora a *Resource* tenha como principal função o tratamento de pedidos HTTP recebidos pelo serviço,

esta não é diretamente dependente do pedido, necessitando apenas de receber os dados necessários ao seu tratamento.

Além disto, esta abordagem orientada a objetos e ao mesmo tempo modular permite que entidades como o *Model* e a *Resource* possam ser utilizadas de modo imperativo pelo programador, facilitando a sua utilização independente de estruturas como o *Router*. Deste modo, além de se facilitar a sua utilização é também conseguido que estas entidades sejam testáveis programaticamente de um modo mais acessível.

Baseado em classes

Os aspetos mencionados anteriormente tais como mapeamentos objeto-relacional e objeto-recurso, a modularidade e separação de responsabilidades, considerados alguns dos fatores mais importantes do desenho desta solução, são conseguidos essencialmente por cada uma dessas entidades se basear em classes.

Ao nível dos *Models*, dado que funcionam como um ORM, este aspeto não é algo fora do comum para este tipo de *frameworks* dado que é já relativamente normal serem usados ORMs bem definidos para o acesso à base de dados, existindo soluções implementadas nas mais variadas linguagens, que usam por base classes ou estruturas de dados relativamente semelhantes.

Ao nível das *Resources* este é um dos pontos que tenta diferenciar esta solução das já existentes. Em grande parte delas, a camada de tratamento de cada um dos pedidos HTTP constituída por um conjunto de métodos independentes, como acontece com o Express, Hapi, Koa, Sails, etc. e, apesar de ser possível isolar os métodos em diferentes módulos dependendo do recurso aos quais estes estão relacionados, os métodos continuam independentes entre si e é necessário associar cada um deles à rota pretendida de modo também independente.

A adoção de uma perspetiva baseada em classes para as *Resources* não é algo completamente novo sendo um conceito que é usado por algumas soluções existentes, como por exemplo o Django REST Framework ou no Rails, no entanto, ao nível de Node.js, esta solução tem tendência em diferenciar-se das já existentes. Este desenho permite que seja possível ao programador definir uma única classe que englobando todos os métodos responsáveis pelo processamento dos pedidos relativos a um determinado do recurso, desde os métodos mais regulares até às ações personalizadas que irão atuar quer sobre a coleção de dados quer sobre um determinado objeto. A existência de uma única entidade proporciona também uma forma fácil para o programador de fazer a sua associação ao *router*, sendo apenas necessário o registo da classe *Resource* criada ao invés de ser necessário associar cada um dos métodos à rota e método HTTP ao qual são destinados.

Métodos Estáticos e de Instância

Além da utilização de classes para a representação dos *Models* e das *Resources*, a forma como os métodos são declarados também possui um significado específico, existindo distinção clara

entre o objetivo dos métodos estáticos e o objetivo da utilização dos métodos de instância das classes.

Relativamente aos métodos estáticos, estes têm apenas o objetivo de aceder e manipular uma determinada coleção de objetos. Fazendo o paralelismo para a arquitetura REST, tal como todos os pedidos HTTP feitos a URLs que sigam o padrão *'/recurso/'* correspondem ao acesso à coleção de objetos, também nas *Resources* os diferentes métodos correspondentes aos diferentes tipos de pedidos a esta rota devem ser métodos estáticos. A única exceção será o método de acesso a um único objeto da coleção, apesar deste tipo de pedidos ser feito através da rota *'/recurso/:id/'*, nas *Resources* será de igual forma representado por um método estático, dado que o objetivo do mesmo será também aceder à coleção de objetos, mas desta vez usando o parâmetro *':id'* como filtro que lhe permite o acesso a apenas um único objeto.

No que diz respeito aos métodos de instância, seguindo uma perspetiva orientada a objetos, são responsáveis por operações que atuam diretamente na manipulação do estado dos objetos.

Este é um dos fatores que é facilitado pelo facto das *Resources* serem representadas por classes, no entanto, mesmo outras soluções em que isso já acontece, como no Django REST Framework, apesar de ser possível definir todos os métodos, de coleção e de instância, diretamente na mesma vista, a abordagem mais utilizada é definição de uma vista para cada um dos contextos de utilização, estático ou de instância.

Com isto, através da solução desenhada é possível que todo o processo de tratamento de pedidos a um recurso seja tratado apenas por uma entidade, reduzindo assim a quantidade de entidades que têm de ser criadas e consecutivamente o número de conceitos com que o programador terá de lidar.

Autenticação e Autorização

Um dos aspetos que a *framework* pretende ter em conta é a segurança dos dados armazenados e, nesse sentido, foi desenhada uma solução para o controlo de acesso aos dados a dois níveis.

Como primeiro nível do controlo de acesso, é possível associar a cada um dos métodos das *Resources* parâmetros indicadores do tipo de mecanismo de autenticação a que os pedidos estarão sujeitos antes que seja procedido ao seu tratamento. Na solução implementada estarão disponíveis alguns mecanismos de autenticação, no entanto, é pretendido dar liberdade ao programador do serviço de implementar o seu próprio mecanismo de autenticação.

Apesar de ser necessária autenticação para aceder a um determinado recurso, por vezes pode ser pretendido restringir ainda mais o acesso a determinados tipos de operações ou dados, onde, determinados tipos de operações podem não ser permitidas a determinados utilizadores atendendo ao papel que estes possuem no sistema.

Com isto surge então necessidade da criação de uma nova camada de controlo de acessos, onde para além de obrigar a que os pedidos passem por um mecanismo de autenticação também verifica a que grupos de permissões é que o utilizador pertence no contexto do serviço.

Este processo é garantido pela presença de dois tipos de modelos de dados já predefinidos na *framework*, os utilizadores e os grupos de permissões, que se relacionam entre si.

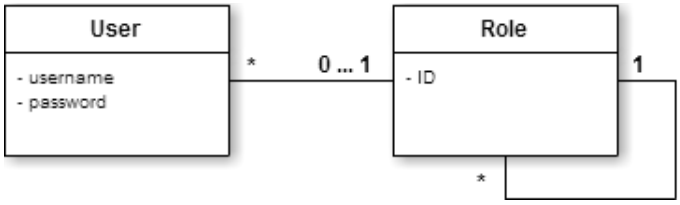

Figura 3.2: Relação entre utilizadores e grupos de permissões

O modelo de dados definido para os grupos de permissões permite que seja estabelecida uma hierarquia entre o próprio modelo, através da qual é possível indicar que um determinado grupo possuirá permissões de acesso a recursos que sejam limitados a grupos de permissões que estejam direta ou indiretamente relacionados e que ao mesmo tempo sejam hierarquicamente inferiores.

Anotações

Tanto a classe *Resource* como a classe *Model* podem ser anotadas/modificadas por forma a que sejam dotadas de propriedades que lhes permitam fazer todo o trabalho a que possam ser sujeitas, tais como a validação dos dados recebidos, introdução de propriedades predefinidas para a consulta de dados e para o formato de saída dos resultados, bem como introdução de propriedades de controlo de acesso tanto ao nível de autenticação como ao nível do controlo de permissões.

Assim, para a definição das *Resources* foi desenhado um conjunto de anotações que podem ser utilizadas:

Name Esta anotação permite que seja associado um identificador à *Resource*, para que, no momento da associação a um *Router*, se não for especificado nenhum caminho, ser utilizado como caminho de acesso à *Resource* no serviço.

Input Através desta anotação é possível indicar qual o esquema de dados que será utilizado para a validação dos dados. Dado tratar-se de apenas uma anotação dá a possibilidade ao programador de, no corpo dos métodos, proceder à validação dos dados de uma forma simples.

Query Permite associar propriedades de consulta de dados predefinidas, que serão apenas utilizadas aquando da consulta de um conjunto de objetos de uma determinada coleção. Pode ser indicada a ordem com que é pretendido que os resultados sejam ordenados, o tamanho da paginação dos mesmos bem como os filtros a aplicar na consulta.

Este é o tipo de propriedades que normalmente é apenas indicado no momento em que é feito o pedido ao recurso, no entanto, esta solução permite que sejam indicados parâmetros predefinidos pelo programador para o acesso à *Resource*.

Output Através deste decorador é possível indicar quais as propriedades às quais os resultados devolvidos pela *Resource* devem obedecer. É possível associar propriedades de projeção, ou seja,

escolha de quais os atributos dos dados é necessário incluir na resposta, mas também indicar quais os objetos associados que devem ser embebidos nas respostas.

Mais uma vez, este é o tipo de propriedades que normalmente é indicado pelo cliente do serviço e não pela especificação do recurso, no entanto, tal como acontece nas propriedades de *Query*, é permitido que sejam indicados parâmetros predefinidos.

Format Além da *framework* permitir negociação de conteúdo, ou seja, permitir ao cliente escolher qual o formato como os resultados dos seus pedidos ao servidor são renderizados, é também permitido associar um formato de renderização à *Resource*. Em vez de funcionar como um formato de renderização predefinido, como acontece nos casos anteriores, a partir do momento em que é associado um formato este passa a ser considerado o formato de renderização que será sempre utilizado, independentemente de ser requisitado um outro formato.

Authentication Um dos objetivos da *framework* é permitir o controlo de acessos às suas *Resources* e uma das formas é obrigar a que os pedidos sejam sujeitos a um processo de autenticação. Deste modo, através desta anotação é possível indicar a necessidade de autenticação para o acesso a uma determinada *Resource* bem como o método de autenticação que deve ser usado.

Roles Esta é o tipo de anotação responsável por um dos níveis do controlo de acessos, permitindo indicar quais os grupos de utilizador autorizados a aceder a uma determinada *Resource*.

Documentation Este tipo de anotação não possui nenhum significado ao nível do tratamento da lógica de tratamento dos pedidos à *Resource* no entanto pretende ser uma forma de documentação da *Resource* ou de cada um dos seus métodos.

Ainda no que diz respeito às *Resources*, existem mais dois tipos de decoradores que podem ser associados, no entanto estes fazem mais sentido aquando da utilização de uma *GenericResource*.

Model Dado a *GenericResource* possuir já a implementação predefinida é necessário que lhe seja associado um *Model* para que esta possa fazer a ligação à base de dados. Como as entidades *Resource* e *Model* possuem uma interface de utilização semelhante, cada um dos métodos na *GenericResource* terá uma correspondência direta aos métodos do *Model* recebido como argumento.

Methods Mais uma vez, tal como acontece para a anotação *Model*, dado que a *GenericResource* possui a implementação de todos os métodos base, é possível que o programador pretenda que apenas alguns desses métodos sejam usados, sendo assim possível indicar quais os métodos que são permitidos.

Tal como nas *Resource* também aos *Models* podem ser aplicados alguns dos tipos de anotações já mencionados, fazendo apenas sentido a utilização dos decoradores *Query*, *Output*, *Input* e *Name*. Das anotações que devem ser utilizadas nos *Models*, praticamente todas funcionam de igual forma ao já mencionado anteriormente, à exceção do *Name*. Neste contexto o decorador *Name* além de funcionar como um identificador para o *Model* ao qual for associado, também representa o nome da tabela da base de dados à qual será feito o acesso aos dados.

Documentação Automática

A solução pretendeu ir ao encontro de uma das boas práticas enunciadas para a este tipo de serviços, disponibilizando uma boa documentação da API e, tendo em conta este objetivo, foi planeada a inclusão de dois mecanismos de documentação automática dos diferentes recursos disponibilizados pelo serviço. Atendendo aos diferentes tipos de clientes possíveis, será disponibilizada uma documentação em formato JSON programaticamente interpretável e outra mais interativa, dotada de uma iterface gráfica com formulários de teste.

No que diz respeito ao primeiro tipo de documentação, a *framework*, de um modo automático, disponibiliza o método HTTP OPTIONS para a generalidade do serviço. Assim, na raiz do serviço, o cliente pode ter acesso ao conjunto de recursos disponibilizados pelo mesmo e a respetivo caminho de acesso a cada um dos recursos. Para cada um dos deles pode também ter acesso ao conjunto de informações relevantes para a sua utilização, tais como os métodos disponíveis e, para cada um deles, o esquema de validação de dados, os parâmetros de *Query* e *Output* predefinidos pela *Resource* e ainda as propriedades relativas ao controlo de acesso dos objetos.

Relativamente ao segundo tipo de documentação enunciado, a solução será dotada de uma interface *Web* onde, para além de serem apresentados todos os *endpoints* disponíveis, existem formulários que permitem a interação e o teste da API, possibilitando assim que qualquer pessoa, sem qualquer conhecimento sobre a implementação do serviço consiga compreender o seu funcionamento e até interagir com o mesmo.

Um dos aspetos importantes que são tidos em conta pela solução desenhada é a baixa intervenção necessária pelo programador do serviço para que esta documentação seja construída. A única forma como este pode interagir com a documentação, se achar necessário, é anotando tanto as *Resources* como cada um dos seus métodos com um objeto de documentação, no qual é indicado um título e uma descrição da entidade à qual se refere, e a *framework* encarregar-se-á de incluir essas propriedades na documentação com todas as outras propriedades já indicadas.

Realtime

Para que através de uma API REST seja possível que os dados presentes do lado do cliente sejam atualizados em tempo real, é necessário que o cliente faça pedidos ao serviço com uma frequência bastante elevada, o que faz com que grande parte desses pedidos sejam desperdiçados

pela inexistência de alterações e que, ao mesmo tempo, provoquem uma elevada carga no serviço e consequentemente uma menor disponibilidade.

De forma a ultrapassar este problema, a solução pensada para esta *framework* permite que seja utilizada uma abordagem de programação reativa, na qual as alterações dos dados são automaticamente propagadas para os seus clientes. Deste modo, fugindo aos moldes tradicionais de uma arquitetura REST, os clientes poderão subscrever um canal de comunicação no qual as alterações dos dados serão automaticamente propagadas, permitindo assim que a disponibilização dos dados em tempo real não seja dependente da frequência dos pedidos.

Esta solução torna possível que, de um modo simples, os utilizadores tenham acesso às alterações dos dados e a sua sincronização entre os diferentes clientes do serviço seja relativamente simples. Esta funcionalidade, que apenas através das especificações de uma arquitetura REST se tornaria mais difícil de alcançar, permite, além de libertar carga do servidor, promover também a performance e a escalabilidade do serviço.

Cache do lado do servidor

A performance dos serviços construídos pretende também ser um dos focos da solução desenhada. Para isso a *framework* possibilita a utilização de mecanismos de *cache* a vários níveis do lado do servidor, por forma a reduzir a carga de trabalho necessária para o tratamento de cada um dos pedidos HTTP recebidos.

Um dos primeiros pontos onde esse mecanismo de *cache* é encaixado na *framework* é na comunicação entre o *Router* e as *Resources*. Neste ponto será colocada uma camada onde é possível armazenar os dados referentes aos diferentes pedidos efetuados e assim, quando for feito um novo pedido, se a *cache* ainda se encontrar válida, serão utilizados esses dados na resposta ao cliente ao invés de efetuar novamente todo o processo de tratamento do pedido.

O outro ponto onde a *cache* é também aplicada é no acesso à base de dados. Dado esta ser uma das etapas que num serviço deste tipo pode levar a um maior impacto a nível de tempo de resposta e consecutivamente na performance do sistema, é também idealizada uma camada de *cache* entre os *Models* e o acesso à base de dados . Assim, mais uma vez, desde que os dados armazenados em *cache* ainda sejam considerados válidos, as *Resources* irão aceder aos dados em *cache* diminuindo assim a quantidade de comunicações necessárias à base de dados, permitindo reduzir o tempo necessário para que o serviço consiga devolver uma resposta aos seus clientes, tornando-se assim mais eficiente.

Models do lado do cliente

O objetivo da existência de documentação do serviço é auxiliar os seus clientes na forma como estes interagem com o mesmo, indicando quais os métodos disponíveis e as propriedades a que cada um deles está sujeito.

Por forma a permitir um auxílio ainda maior ao programador, esta solução idealiza também a possibilidade do serviço ter a capacidade de geração de código de cliente, em JavaScript. Assim,

em vez de os clientes necessitarem de aceder à documentação do serviço e implementarem eles próprios uma forma de acesso ao mesmo obedecendo às propriedades associadas a cada *Resource*, o próprio serviço tem a capacidade de gerar o código de cliente necessário para que os programadores das aplicações cliente possam utilizar como forma de abstrair o acesso ao serviço.

Aliado à documentação e à facilidade no acesso ao serviço, esta funcionalidade permite a utilização de entidades semelhantes aos *Models* utilizados do lado do serviço mas desta vez do lado do cliente reduzindo assim a necessidade de assimilação de novos conceitos para os programadores que implementem tanto a componente de servidor como a componente de cliente do sistema.

O desenho da utilização de *Models* do lado do cliente permite que outras funcionalidades possam ser acrescentadas à *framework*, desta vez com foco na utilização do serviço do lado do cliente.

Um objetivos da *framework* é permitir a utilização de uma abordagem de programação orientada à reatividade e, deste modo, através da utilização de *Models* do lado do cliente, é possível abstrair essa tarefa do programador. Em vez de ser necessário ao programador fazer a subscrição manual ao canal de comunicação usado para a transmissão dos eventos, os *Models*, oferecem eles próprios a possibilidade de fazer essa subscrição de uma forma relativamente simples para o programador. A única coisa que o programador terá que fazer aquando da requisição de um objeto é indicar que pretende receber os eventos relacionados com o mesmo, e assim, qualquer alteração será automaticamente transmitida ao cliente e os dados atualizados de acordo com o tipo de evento recebido.

Tal como na componente de servidor podem ser utilizados mecanismos de *cache*, a utilização deste tipo de mecanismos do lado do cliente é também uma forma de abrir a possibilidade de integração da *framework* numa aplicação desenvolvida numa abordagem *offline-first*, na qual os dados são guardados localmente, promovendo não só a performance mas também a utilização das aplicações com um mínimo de necessidade de conexão. Deste modo, tanto o alojamento como a manipulação dos dados é feita localmente e apenas se for conseguida conexão ao serviço é feita a sincronização dos dados, mas de uma forma completamente transparente aos seus utilizadores.

Capítulo 4

Implementação

Tendo em conta o desenho da solução apresentado no capítulo anterior, neste capítulo será relatado seu o processo de implementação. Para isso haverá um foco na enumeração das principais tecnologias utilizadas para a concretização da solução bem como apresentadas as funcionalidades da solução que realmente foram implementadas. Além disso, este capítulo servirá também como meio de clarificação do modo como as diferentes componentes e funcionalidades da solução foram desenvolvidas, focando assim não só os detalhes de implementação mas também as diferentes decisões tomadas.

4.1 Tecnologias Utilizadas

Relativamente às tecnologias utilizadas, são apresentadas em primeiro lugar a linguagem utilizada, bem como as principais características que permitiram a implementação da solução desenhada. Além disso são também apresentadas outras tecnologias tais como os diferentes tipos transpiladores de código utilizados e o seu impacto na implementação.

Por último é apresentado o tipo de base de dados utilizado bem como as principais caraterísticas que levaram a que fosse escolhido.

4.1.1 JavaScript

A adoção das tecnologias base para a concretização desta dissertação não foi alvo de escolha, dado que o principal foco desta dissertação foi o desenho e implementação de uma solução suportada pela linguagem de programação JavaScript e plataforma Node.js para o desenvolvimento de serviços *web*. No entanto, alvo de estudo e de escolha foram as funcionalidades e conceitos desta linguagem, tendo havido uma especial atenção na adoção das principais tendências e novidades relativas à linguagem JavaScript.

Ao nível das funcionalidades mais recentes do JavaScript, um dos objetivos para a implementação desta solução passou pela utilização das mais recentes características introduzidas pela especificação ES6, tais como classes, *arrows*, *template strings*, *modules* e *promises*, etc., e ainda algumas novidades da especificação ES7.

Algumas das caraterísticas utilizadas serviram, de certa forma, apenas para aumentar a produtividade e o desenvolvimento de uma solução o mais moderna possível. No entanto aspetos como classes e *promises* foram fundamentais no desenho e implementação da *framework*.

Em primeiro lugar, praticamente toda a solução foi implementada através de uma abordagem orientada a objetos e, apesar de já ser possível fazer a implementação de classes através da sintaxe antiga de JavaScript, a introdução de classes nesta nova especificação permitiu fazer a implementação das diferentes entidades de uma forma mais declarativa e ao mesmo tempo oferecendo as mesmas funcionalidades da sintaxe anterior de um modo mais simples.

Relativamente às *promises*, estas também foram usadas praticamente em toda a implementação da *framework* na execução de tarefas assíncronas. A sua utilização foi, de modo geral, acompanhado de uma nova caraterística da proposta de especificação ES7, as funções assíncronas, que permitiu em muito melhorar a legibilidade do código produzido e, acima de tudo, simplificar a utilização de *promises*, no sentido em que ao invés de ser necessário interagir com os seus *handlers*, apenas declarando a função como assíncrona é possível esperar pelo seu resultado de uma forma semelhante ao que acontece com as funções síncronas.

Além das caraterísticas já mencionadas, os decoradores, introduzidos também pela proposta de especificação ES7, foram uma das caraterísticas que permitiram a implementação da solução desenhada de uma forma relativamente simples e amigável para os futuros utilizadores. Esta característica permitiu fazer a anotação dos diferentes tipos de entidades da aplicação, dotando-as dos diferentes tipos de propriedades necessárias ao correto funcionamento.

4.1.2 Transpiladores de Javascript

Apesar de as funcionalidades utilizadas referentes à especificação ES6 e à proposta de especificação ES7 poderem ser usadas neste momento pela linguagem JavaScript, estas ainda não se encontram totalmente implementadas de forma nativa e, por isso, ainda é necessário a utilização de transpiladores de código que permitam transformar o código em ES6/ES7 para uma sintaxe compatível com ES5 e possível de ser executada em qualquer ambiente. Neste sentido, durante a implementação da solução foram utilizados dois transpiladores diferentes, o *Traceur* e o *Babel*.

Num ponto mais inicial da implementação da *framework* foi utilizado o transpilador *Traceur*, por forma a permitir a utilização de código ES6 na implementação da solução no entanto, foram verificadas algumas inconveniências na utilização do *Traceur*. Em primeiro lugar, esta ferramenta, além de obrigar a que o código ES6 fosse transformado, necessitava que fosse incluído no código da *framework* um *script* que seria executado em tempo de execução auxiliando no processo de transformação também durante a execução da aplicação.

Relativamente ao código gerado, a utilização desta ferramenta originou também alguns problemas. Durante da utilização do *Traceur* não foi implementar módulos tal como é descrito na especificação ES6, obrigando a usar a sintaxe antiga, e ainda, outro ponto desvantajoso encontrado na utilização desta ferramenta foi a falta de controlo da forma como as anotações são adicionadas às entidades de destino.

Perante estes factos, a utilização do *Traceur* foi abandonada e adotado o *Babel*. Através do *Babel*, além de ter sido facilitada a utilização de todas as funcionalidades tanto da especificação ES6 e proposta de especificação ES7, permitiu que não fosse necessário adicionar *scripts* para a execução da *framework*, sendo apenas necessário executar uma tarefa de compilação antes da execução do código fonte. A utilização do *Babel* veio ainda permitir a utilização dos decoradores de JavaScript de acordo com a proposta de especificação existente, o que não acontecia com o *Traceur*, havendo um maior controlo sobre a forma como estes foram aplicados, tanto apenas anotando as entidades alvo como facilitando a alteração no comportamento das mesmas.

Por último mas igualmente importante, o *Babel* permitiu que, ao contrário do que acontecia com o *Traceur*, o código gerado fosse mais facilmente lido e compreendido, facilitando assim o processo de desenvolvimento e de depuração da *framework*.

4.1.3 Base de dados

Para o armazenamento dos dados, embora a solução tenha como objetivo permitir a utilização de vários tipos de bases de dados, a solução implementada contemplou apenas a utilização de RethinkDB como sistema de base de dados para a *framework*. Foi ainda considerada a hipótese de utilização de MongoDB, no entanto, algumas caraterísticas do RethinkDB fizeram com que esta tenha sido a opção tomada.

Em primeiro lugar, o controlador de Node.js, permite que as consultas à base de dados sejam feitas através do encadeamento das diferentes opções da consulta, permitindo assim a construção de um código relativamente limpo evitando pirâmides de código. A interface de gestão da base de dados, é bastante amigável, permitindo o acesso a operações de replicação e de fragmentação da base de dados. Este é um dos principais pontos vantajosos desta base de dados quando comparada com MongoDB, que não possui uma página de administração geral.

A adoção de RethinkDB foi também influenciada por este tipo de base de dados possuir, implementado de modo nativo, funcionalidades que permitem a implementação de aplicações com uma abordagem de reatividade. A funcionalidade *changefeeds* permite que os clientes da base de dados subscrevam as alterações tanto de tabela, como de um documento ou mesmo de uma consulta e de cada vez que esse um determinado evento ocorrer estes são notificados. Atendendo que um dos objetivos da solução desenhada considera a possibilidade de utilização da *framework* para a transmissão de dados em tempo real, o RethinkDB foi considerada uma boa opção para auxiliar na implementação desta funcionalidade.

4.2 Metodologia

Como forma de implementação da solução foi seguida a metodologia, representada na figura 4.1, que englobou todo o processo desde a estruturação das narrativas de utilização em diversos módulos até à integração do sistema completo.

Após o desenho da solução, para cada um dos módulos e funcionalidades implementadas foram analisadas algumas soluções já existentes para sua concretização, sendo adotada a melhor

das soluções encontradas, ou no caso em que as soluções encontradas não se enquadravam com a realidade da *framework*, foi desenhado e implementado um novo módulo e integrado com a solução.

Além disto, após a implementação ocorreu o teste das funcionalidades e módulos sendo verificadas se todas as necessidades e requisitos foram cumpridos. Só após esta fase, a componente implementada foi integrada na *framework* de modo a ser possível oferecer as funcionalidades relacionadas aos programadores.

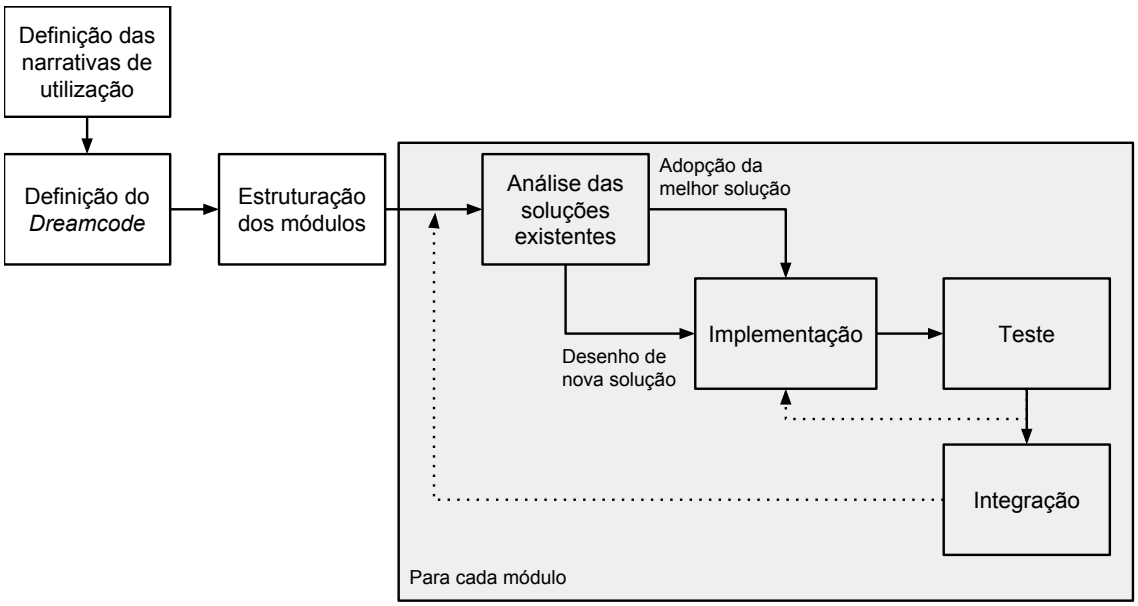

Figura 4.1: Metodologia de implementação da solução

4.3 Funcionalidades Implementadas

Embora tenha havido o desenho de uma solução bastante vasta envolvendo os vários níveis da *framework*, tanto no que diz respeito à componente de servidor como ao nível da componente cliente, a solução implementada não refletiu a totalidade da solução projetada.

Nesta fase de implementação o principal foco foi o desenvolvimento de uma solução que contemplasse as principais funcionalidades relacionadas com a componente de servidor, ou seja, o núcleo central da *framework* capaz de colocar em funcionamento um serviço REST.

Além da implementação de uma solução orientada a objetos, capaz de mapeamentos objeto-recurso e objeto-relacional, agnóstica tanto de base de dados como de *middleware* de tratamento de pedidos HTTP, a solução implementada, graças às mais recentes novidades tecnológicas relacionadas com a linguagem de programação utilizada, permite ainda que seja possível dotar as diferentes entidades como *Models* e *Resources* dos diferentes tipos de propriedades necessárias ao seu correto funcionamento de uma forma simples e programaticamente limpa.

No que diz respeito à documentação, nesta fase foram também integrados na *framework* os dois tipos de documentação projetados, permitindo assim oferecer aos seus clientes uma interface que lhes possibilite não só uma compreensão do serviço de uma forma praticamente independente da implementação mas também o teste do mesmo de uma forma interativa.

Relativamente às funcionalidades como a geração automática de *Models* que além de permitirem uma integração facilitada com o serviço no lado do cliente também permitem a construção de aplicações atendendo a um paradigma de programação reativo e ao mesmo tempo adotando uma abordagem *offline-first*, apesar de estas terem sido tomadas em conta na fase de conceção da solução, foi tomada a decisão de não serem implementadas nesta fase da dissertação.

A decisão de quais as funcionalidades a implementar teve em conta, acima de tudo, a extensão da solução desenhada e a capacidade da sua concretização nos prazos estabelecidos para esta dissertação. Além disso, uma das principais preocupações foi a implementação de uma base estável que permita a criação de serviços REST capazes de serem utilizados e que ao mesmo tempo possam estar preparados para a implementação das funcionalidades que nesta fase ficaram por implementar.

4.4 Detalhes de implementação

Nesta secção são apresentados os principais detalhes da implementação da solução desenhada. Deste modo, em primeiro lugar é apresentado na figura 4.2 o conjunto das entidades implementadas bem como as forma como estas se relacionam entre si.

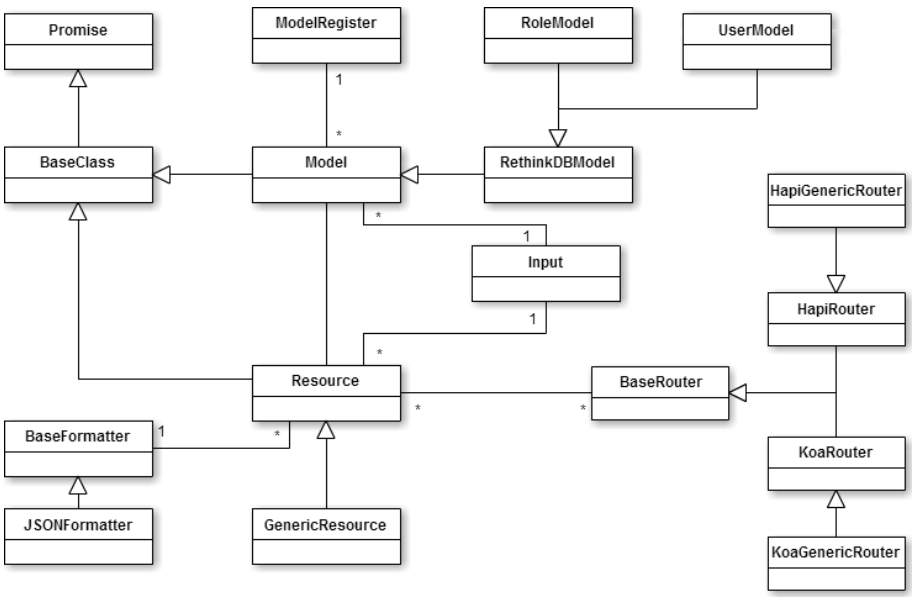

Figura 4.2: Organização das entidades implementadas

4.4.1 *Resources*

Tal como foi desenhado, as *Resources* são entidades que foram implementadas através de classes de JavaScript, permitindo assim encapsular os diferentes métodos disponibilizados pelo recurso em uma só entidade. Para o programador, a definição de uma nova *Resource* será possibilitada através da construção de uma nova classe que herde da classe *Resource*.

```
1  class MyResource extends Resource{
2      constructor(){
3          super(async function () {});
4      }
5      static async fetch(){}
6      static async fetchOne(){}
7      async put(){}
8      async patch(){}
9      async delete(){}
10 }
```

Listing 4.1: Definição de uma classe *Resource*

Além do significado dos métodos estáticos e de instância, o nome dos métodos também possui um significado especial, correspondendo exatamente ao método HTTP ao qual estarão associados, de modo a facilitar a compreensão do objetivo de cada um dos métodos. As únicas exceções à regra introduzidas foram o construtor da classe e os métodos *fetch* e *fetchOne*.

Ambos os métodos *fetch* e *fetchOne* correspondem aos métodos HTTP GET para os casos em que a intenção será aceder a um conjunto de objetos ou apenas a um dos elementos da coleção, filtrado por identificador, mas pelo facto de ambos serem considerados métodos estáticos foi adotada outra nomenclatura para a distinção dos dois métodos.

No que diz respeito ao construtor, este método permite a criação de novos objetos e a sua inserção nas coleções de dados, e corresponde a um método HTTP POST sobre o recurso. A implementação deste método atendendo ao objetivo proposto foi um dos desafios de implementação, tanto nas *Resources* como nos *Models*, dado que por omissão não é possível que um construtor faça invocações a métodos assíncronos. Como forma de ultrapassar esta dificuldade, estas entidades passaram a ser herdeiras de uma classe de *Promise*. Assim, no construtor, todo o código assíncrono será passado ao construtor da classe herdada e deste modo é possível usar este método de igual forma como qualquer outro dos métodos da classe.

```
1  class MyResource extends Resource {
2      constructor(options = {}) {
3          super(async () =>  { /* async code  */ });
4      }
5  }
```

Listing 4.2: Implementação do construtor da classe *Resource*

Decoradores

Para o processo de anotação/modificação do comportamento dos diferentes métodos de entidades como as *Resources*, à semelhança do que acontece com outras linguagens como, por exemplo, Java ou Python, foram utilizados decoradores.

Apesar deste tipo de abordagem ainda não ser muito usada pela comunidade de desenvolvimento de JavaScript, as mais recentes novidades, em termos de proposta de especificação tornam esta abordagem numa realidade mais facilmente alcançável, sendo facilitada a anotação ou até a alteração do comportamento dos métodos das classes.

Como pode ser visto no código apresentado em 4.3, o processo de anotação poderia ter sido alcançado sem a necessidade de utilização de decoradores, no entanto, a sua utilização permite que a anotação das diferentes entidades seja feita de um modo mais amigável e controlado e ao mesmo tempo permitindo englobar toda a lógica dentro do escopo da classe responsável.

```
1  //annotation with decorators
2  @MyAnnotation({key:"value"})
3  class MyResource extends Resource{
4      @MyAnnotation({key:"value"})
5      static async fetchOne(options){/*...*/}
6  }
7
8  //annotation with traditional approach
9  class MyResource extends Resource{
10   static async fetchOne(options){/*...*/}
11 }
12 MyResource.myAnnotation = {key:"value"}
13 MyResource.fetchOne = myAnnotation = {key:"value"}
```

Listing 4.3: Anotação através de decoradores ou através da abordagem tradicional

Apesar de os programadores terem a possibilidade de não utilizarem decoradores e proceder à anotação das diferentes entidades através da abordagem tradicional, é preferível que sejam utilizados decoradores como uma forma de proporcionar entidades o mais isoladas possível. Além disso, foram disponibilizados um conjunto de decoradores pré-definidos que podem ser utilizados pelas *Resources* atendendo aos diferentes tipos de anotações necessárias, possibilitando assim a anotação através de um ambiente mais controlado para os programadores.

@Name

Este é um dos decoradores mais simples, apenas recebe um identificador que poderá posteriormente ser usado como caminho de acesso à *Resource* através da API.

```
1  @Name("myResourceIdentifier")
2  class MyResource extends Resource{}
```

Listing 4.4: Utilização do decorador *Name*

@Input

Este decorador recebe o objeto responsável pela validação dos dados recebidos, ou seja, recebe uma instância da classe *Input*, contendo todas as propriedades de validação dos dados.

```
@Input(inputObject)
class MyResource extends Resource{}
```

Listing 4.5: Utilização do decorador *Input*

@Output

Através deste decorador são indicadas as propriedades relativas à forma como os resultados serão devolvidos:

- ***_fields***: deve ser indicado um *array* com o nome dos atributos a incluir nos resultados.
- ***_embedded***: deve ser indicado um *array* com os atributos que devem incluir os objetos relacionados de forma embebida.

```
@Output({
    _fields: ["name","categories"],
    _embedded: ["categories"]
})
class MyResource extends Resource{}
```

Listing 4.6: Utilização do decorador *Output*

@Query

Este decorador deve ser utilizado apenas para a transmissão das propriedades relativas a consultas de conjuntos de objetos:

- ***_sort***: um *array* com os atributos que devem ser utilizados na ordenação dos resultados. A forma como os atributos são indicados permite também incluir a ordem com que a ordenação é efetuada para cada um dos atributos.
- ***_filter***: um objeto com os valores a que os objetos retornados devem obedecer.
- ***_page_size***: o número de resultados que devem ser incluídos na devolução de conjuntos de objetos.

```
@Query({
    _sort: ["-name","address"],
    _filter: {type : "grill"},
    _page_size: 15
```

```
5 })
6 class MyResource extends Resource{}
```

Listing 4.7: Utilização do decorador *Query*

@Format

Deve ser indicada uma subclasse de *BaseFormatter*, a qual será responsável pelo formato de renderização dos resultados

```
1 @Format(JSONFormat)
2 class MyResource extends Resource{}
```

Listing 4.8: Utilização do decorador *Format*

Até neste ponto a *framework* foi implementada por forma a dar ao programador a liberdade de ele próprio implementar a sua forma de renderização de resultados necessitando apenas de implementar a sua própria subclasse.

```
1 @MediaType('application/text')
2 class TextFormat extends BaseFormatter{
3     static format(data){
4         return data.toString();
5     }
6 }
```

Listing 4.9: Utilização do decorador *Query*

@Authentication

Através deste decorador deve ser indicado o tipo de autenticação que deve ser utilizado para aceder à *Resource* em questão.

```
1 @Authentication('basic')
2 class MyResource extends Resource{}
```

Listing 4.10: Utilização do decorador *Authentication*

@Roles

Atendendo ao controlo de acessos, este decorador permite indicar ao sistema, através de um *array* com os identificadores dos grupos de utilizadores, quais os papéis que os utilizadores devem possuir no sistema para que possuam autorização para utilizar os métodos da *Resource*.

```
1 @Roles(['client', 'editor'])
2 class MyResource extends Resource{}
```

Listing 4.11: Utilização do decorador *Roles*

@Documentation

Este decorador permite ao programador documentar a *Resource* implementada bem como cada um dos seus métodos. Podem ser usados outros atributos, no entanto, na documentação interativa apenas são associados o título e a descrição. Assim, todos os restantes dados associados a este decorador apenas estarão disponíveis através da interface HTTP OPTIONS.

```
1 @Documentation({
2     title : "Resource title",
3     description : "Resource description"
4 })
5 class MyResource extends Resource{}
```

Listing 4.12: Utilização do decorador *Documentation*

Ações

A *framework* oferece ainda a possibilidade de implementação de ações que são disponibilizadas pela API, podendo também essas ações ser aplicadas num contexto estático ou de instância e disponibilizadas respetivamente através de endereços que sigam os seguintes padrões:

- ***/<nome_recurso>/<nome_ação>***: para as ações estáticas
- ***/<nome_recurso>/<id>/<nome_ação>*** para as ações de instância

Para a utilização de ações, de ambos os tipos, os programadores apenas necessitam de declarar um novo método na *Resource* em questão e implementar a lógica pretendida no método. Para que um determinado método seja encarado como uma ação é necessário que, em primeiro lugar seja anotado através do decorador ***@Action()***, e em segundo lugar o nome do método obedeça ao padrão *<método_HTTP>_<nome_ação>*. A primeira implementação consistia em que todos os métodos cujo nome seguisse esse padrão fossem automaticamente considerados ações, no entanto, por forma a facilitar a distinção entre os diferentes métodos auxiliares das classes e aqueles que representam ações foi adotada a opção de adicionar um decorador e assim restringir as ações apenas aos métodos anotados.

A introdução da possibilidade da implementação de ações vai ao encontro da necessidade de efetuar operações que não são consideradas *RESTful* mas que normalmente são necessárias para a introdução de funcionalidades extra nos recursos. O mais habitual é este tipo de operações serem requisitadas através do método HTTP POST, no entanto esta solução pretende oferecer aos programadores a possibilidade destes implementarem ações para qualquer um dos métodos HTTP, apesar de o nome da ação ser exatamente o mesmo. É neste sentido que surgiu a abordagem mencionada para a definição do nome das ações, oferecendo assim a possibilidade de definir vários métodos HTTP para a mesma ação.

```
class MyResource extends Resource{
    @Action()
    static async get_schema(){
        return MyResource.getSchema();
    }
    @Action()
    static async post_schema(){
        return MyResource.getSchema();
    }
}
```

Listing 4.13: Implementação da ação *schema* para diferentes métodos HTTP

GenericResource

Por forma de evitar que o programador seja obrigado a implementar a sua própria lógica para cada um dos métodos das *Resources* foi criada uma subclasse, a *GenericResource* que oferece já a implementação regular para os métodos básicos.

@Model

Para que cada um dos métodos seja disponibilizado, é necessário que exista um *Model* capaz de fazer a conexão à base de dados, e assim, uma das formas como pode ser associado é através deste decorador. Como a interface das duas entidades é semelhante há uma correspondência entre cada um dos métodos da *Resource* e os métodos do *Model* que deve ser invocado por cada um deles.

new GenericResource() -> new Model()
GenericResource.fetch() -> Model.fetch()
GenericResource.fetchOne() -> Model.fetchOne()
(...)

```
@Model(MyModelClass)
class MyResource extends GenericResource{}
```

Listing 4.14: Utilização do decorador *Model*

@Methods

Para os casos em que o programador utilize uma *GenericResource* mas não pretende disponibilizar todos os seus métodos pode fazer uma restrição através deste decorador indicando os nomes dos métodos a autorizar que sejam utilizados.

De forma a efetuar essa restrição de um modo coerente, os identificadores dos métodos são o seu nome, ou no caso de serem métodos estáticos, possuem o prefixo "static"evitando assim confusão de interpretação entre métodos com o mesmo nome mas com contextos diferentes.

```
@Methods(["constructor", "put", "static.fetchOne", "static.fetch"])
class MyResource extends GenericResource{}
```

Listing 4.15: Utilização do decorador *Methods*

Além disso, o programador pode pretender alterar a implementação de um determinado método ou até proceder à sua anotação, necessitando para isso apenas de fazer *override* aos métodos pretendidos.

```
class MyResource extends Resource{
    static async fetchOne(){
        return await new MyResource();
    }
    @Roles["admin"]
    async delete(){
        return await super.delete();
    }
}
```

Listing 4.16: Alterar o comportamento de uma *GenericResource*

4.4.2 *Models*

À semelhança do que acontece com as *Resource* também os *Models* foram implementados através de classes de JavaScript. Dado que os *Models* possuem como objetivo o acesso e disponibilização para outras entidades dos dados existentes na base de dados estes oferecem uma interface que permite a manipulação dos dados também seguindo uma lógica orientada a objetos. Além disso, praticamente todos os pormenores de implementação seguem as diretivas atendidas na implementação das *Resources* tanto ao nível da lógica dos métodos das *Resources*, do seu contexto estático e de instância, a nomenclatura utilizada, oferecendo assim uma interface semelhante e ainda uma forma de permitir um mapeamento direto entre as duas entidades.

```
class RethinkDBModel extends Model{
    constructor(){
        super(async function () {});
    }
    static async fetch(){}
    static async fetchOne(){}
    async put(){}
    async patch(){}
    async delete(){}
}
```

Listing 4.17: Definição de uma classe *Model*

Na *framework* foi implementada uma subclasse que permitisse o acesso a bases de dados RethinkDB, no entanto, o sistema encontra-se desenhado de modo a que para que o acesso a outros tipos de bases de dados seja possível é apenas sendo necessário fazer a implementação de uma subclasse cujos métodos acedem à base de dados pretendida.

Tal como as *Resource* também os *Models* podem ser decorados, fazendo apenas sentido a utilização dos decoradores ***@Query, @Ouput, @Input*** e ***@Name***, no entanto, apenas este último possui um significado um pouco diferente. Este decorador irá servir não só como identificador do *Model* mas também representa o nome da tabela da base de dados com a qual irá interagir.

Como foi mencionado, a *framework* permite que em vários níveis as entidades sejam anotadas com propriedades do mesmo tipo, no entanto, no caso das propriedades de *Query* e de *Output* terá de haver uma consolidação dessas propriedades de modo a que o acesso e a disponibilização dos dados obedeça às propriedades pré-definidas recebidas tanto pelos *Models* e *Resources* mas também por aquelas que são recebidas em cada pedido.

Para que essa consolidação fosse possível foram estabelecidas um conjunto de regras para cada uma das propriedades:

- ***_sort:*** aquelas que forem indicadas numa camada superior são sobrepostas às anteriores, respeitando a ordem pela qual os pedidos são tratados, ou seja, pedidos, *Resources* e *Models*.
- ***_page_size:*** o tamanho da paginação será o valor mínimo encontrado nas diferentes camadas.
- ***_filter:*** os filtros aplicados seja um conjunto dos filtros indicados nas diferentes camadas, sendo o resultado obrigado a cumprir todos eles.
- ***_embedded:*** os atributos que serão embebidos serão aqueles que forem comuns a todas as camadas. No caso de não ser indicada esta propriedade é assumido que são aceites todos os atributos.
- ***_fields:*** os atributos que serão incluídos nos resultados serão aqueles que, mais uma vez, forem comuns a todas as camadas. Também para esta propriedade é assumido que por predefinição são aceites todos os atributos.

4.4.3 Router

Ao nível do *Router*, este foi implementado de modo a proporcionar uma adaptação fácil a qualquer tipo de *middleware* de tratamento de pedidos HTTP no entanto no contexto desta dissertação foram implementados apenas dois tipos de *Routers*, através da *microframework* Koa e Hapi.

Para a implementação dos *Routers* foi criada uma outra classe, a *BaseRouter* onde foram implementados os métodos necessários à construção do serviço e ao tratamento dos pedidos que são semelhantes e necessários independentemente do tipo de *middleware* usado. Assim, para os

diferentes tipos de *Routers* é apenas necessário criar uma subclasse de *BaseRouter* e implementar os métodos específicos de cada tipo de *middleware* usado.

Durante a implementação foram notadas algumas limitações neste tipo de abordagem, devido ao tipo de funcionamento do *middleware* utilizado, havendo bastantes diferenças na implementação para que o resultado produzido pelos diferentes *Routers* fosse semelhante. Funcionalidades como a autenticação, autorização ou mesmo a integração de módulos de documentação do serviço dependem do *middleware* usado, e por isso, obriga a um trabalho adicional na definição de novos tipos de *Routers*.

Quanto à interface implementada e disponível para o programador esta é relativamente simples, permitindo apenas a associação de *Resources* e após isso a inicialização do serviço.

```
let router = new HapiRouter();
router.register([{
        resource: RestaurantResource
    },
    {
        path: 'addresses',
        resource: AddressResource
    }
])
router.start({port:8080});
```

Listing 4.18: Registo de *Resources* a um *Router*

GenericRouter

Em adição aos tipos de *Routers* básicos, foram também implementados *GenericRouters* que têm como objetivo dotar um serviço de uma interface *NoBackend*.

Através deste tipo de *Router* é possível, de igual modo, fazer a associação de *Resources* à API no entanto também permite que sejam adicionadas *Resources* em tempo de execução de acordo com o tipo de pedidos recebidos no serviço. Assim, a partir do momento em que é feito um pedido HTTP POST bem sucedido a um recurso até ao momento inexistente é criado no serviço um novo tipo de *GenericResource* e a partir desse momento podem ser utilizados todos os métodos básicos para o recurso em questão.

Deste modo, um *Router* básico difere de um *GenericRouter* na medida em que para o mesmo pedido podem ser encontrados diferentes resultados. Como exemplo, para o pedido a seguir mencionado podem ser encontrados diferentes resultados.

```
curl -X POST \
  -d '{ "phone": "+351 222 222 222"} }'\
  https://api.apey-eye.com/phone/
```

Listing 4.19: Pedido POST efetuado a um *Router* e a um *GenericRouter*

Enquanto através da utilização de um *Router* a resposta dada pelo serviço aos seus clientes seria *Status: 404 Not Found*, através da utilização de um *GenericRouter* um novo recurso seria criado.

```
Status: 201 Created
'{
    "id": "e9a092af63654414227f"
    "phone": "+351 222 222 222"
 }'
```

Listing 4.20: Resposta a um pedido POST efetuado a um *GenericRouter*

Todo este processo é possível devido à utilização de *GenericResources*. Apenas através da indicação de um *Name*, a *Resource* construída encarrega-se de verificar se existe um modelo de dados para o tratamento dos respetivos pedidos e, no caso de este não existir, a entidade vai automaticamente associar um novo modelo de dados para o recurso em questão. Por sua vez, esse modelo de dados irá verificar a existência da tabela na base de dados e se necessário procederá à sua criação.

```
@Name('resourceName')
class MyResource extends GenericResource{}
```

Listing 4.21: Utilização de uma *GenericResource*

4.4.4 Validação de dados

Tal como já foi mencionado, a validação dos dados deverá ser realizada através da utilização de um objeto pertencente à entidade *Input*, através do qual são indicados quais os atributos que podem ser encontrados nos dados bem como as propriedades a que cada atributo deve obedecer.

```
let restaurantInput = new Input({
    name:       {type: "string", required: true},
    year:       {type: "number", valid: yearValidator},
    created:    {type: "date", default: "now"},
    type:       {type: "string", choices: [(...)]},
    phone:      {type: "string", regex: ^[0-9]}
});

let yearValidator = function(value){
  if(value <= 2015) return true;
  else throw new Error("Invalid year.");
};
```

Listing 4.22: Definição de um esquema de dados

Para cada um dos atributos do esquema de dados podem ser utilizadas as propriedades:

- *type*: o tipo de dados a que o valor deve pertencer.
- *required*: a obrigatoriedade de existência de um valor para o atributo em questão
- *regex*: expressão regular à qual o valor deve obedecer.
- *valid*: uma função que permite validar o valor do atributo.
- *choices*: um conjunto de valores ao qual o valor do atributo deve pertencer.
- *default*: um valor pré-definido que deve ser associado ao atributo no caso de este não possuir nenhum valor associado.

Assim, após a associação a *Resources* ou *Models*, estes objetos podem ser direta ou indiretamente usados para a validação dos dados recebidos pelos diferentes métodos.

```
1  @Input(inputObj)
2  class MyResource extends Resource{
3      async put(options){
4          await MyResource.valid(options.data)
5      }
6  }
7  @Input(inputObj)
8  class MyModel extends RethinkDBModel{
9      async put(options){
10         await MyModel.valid(options.data)
11     }
12 }
```

Listing 4.23: Validação de dados

Além das funcionalidades consideradas básicas no que diz respeito à validação dos dados, esta é a entidade que também é responsável pela definição de relações entre os diferentes modelos de dados. Para isso, foi implementada uma estrutura de propriedades para os *Inputs*, que permite fazer a definição de três tipos de relações, e para cada uma delas surgiram também outras propriedades que devem ser indicadas:

- *model*: o identificador do modelo de dados com o qual o esquema se relaciona.
- *inverse*: o nome do atributo responsável pela relação no modelo de dados de destino.
- *through*: para um dos tipos de relações implementadas é necessária a existência de uma modelo de dados intermédio, o qual deve ser indicado neste campo.

Este esquema além de permitir a definição das relações, tal como é o seu maior objetivo, permite também validar os dados de entrada de um ponto de vista relacional, ou seja, além de serem verificadas as propriedades acerca dos valores dos atributos básicos, as relações são de igual forma validadas, havendo uma verificação da existência de cada um dos objetos referenciados.

4.4.5 Relações

Ao contrário do que acontece com grande parte das soluções existentes em que as relações entre os diferentes esquemas de dados são replicados diretamente na definição do esquema de base de dados, nesta *framework* as relações foram implementadas de forma independente da base de dados e apenas com base na definição do esquema de dados que será associado aos *Models* da aplicação. Relativamente às soluções já existentes, esta abordagem possui a vantagem de possibilitar a criação de relações entre os diferentes tipos de *Models* independentemente do tipo de base de dados com que os mesmos interagem.

Para o cumprimento desta funcionalidade foram implementados três tipos de relações:

- **Referência**: um objeto possui uma relação com apenas um outro objeto pertencente ao modelo de dados alvo.

```
var restaurantInput = new Input({
    (...)
    phone:{type: "reference", model:"phone"}
});

var phonesInput = new Input({
    phone_number: {type: "string", required: true}
});
```

Listing 4.24: Definição de uma relação de referência

- **Coleção**: um objeto possui uma relação com um conjunto de outros objetos pertencentes ao modelo de dados alvo.

```
var restaurantInput = new Input({
    (...)
    addresses:{type: "collection", model:"address", inverse:"restaurant"}
});

var addressesInput = new Input({
    address: {type: "string", required: true},
    restaurant: {type:"reference", model:"restaurant"}
});
```

Listing 4.25: Definição de uma relação de coleção

- **Muitos para muitos**: um conjunto de objetos está relacionado com um conjunto de outros objetos pertencentes ao modelo de dados alvo.

```
var restaurantInput = new Input({
    (...)
    categories:{type: "manyToMany", model:"category", inverse:"restaurant"
        , through:"categoryRestaurant"}
});
var categoryInput = new Input({
    name: {type: "string", required: true},
    restaurants: {type: "manyToMany", model: "restaurant", inverse: "
        categories", through: "categoryRestaurant"}
});

var categoryRestaurantInput = new Input({
    category: {type:"reference", model:"category"},
    restaurant: {type:"reference", model:"restaurant"}
});
```

Listing 4.26: Definição de uma relação de muitos para muitos

Apesar das relações serem representadas através dos *Inputs*, o facto de na relação ser indicado o nome do *Model* com o qual irá ser feita a ligação obriga a que esses *Models* realmente sejam implementados e o seu identificador corresponda ao mesmo nome que foi indicado na relação. Este pormenor é bastante importante porque, apesar das entidades parecerem desligadas entre si, a *framework* permite que, apenas através do nome do *Model*, seja possível ter acesso à sua classe e aí efetuar todas as operações que sejam necessárias.

Isto é possível graças a um objeto *singleton*, o *ModelRegister*, que permite registo automático das classes *Model* às quais seja associado um identificador através do decorador *@Name*. Assim, como os esquemas de validação de dados possuem o identificador do *Model* ao qual as relações se referem estes conseguem facilmente aceder à classe responsável e assim utilizá-la por forma a verificar a existência dos objetos referenciados.

Este pormenor de implementação veio também possibilitar que de uma forma fácil, em vez de o valor das relações corresponderem apenas ao identificador de um objeto já existente, seja possível que esse valor represente um novo objeto que, por sua vez, será adicionado à base de dados através do *Model* correspondente e, com as respetivas relações devidamente preenchidas.

Assim é possível que, para a inserção de um objeto que possui outros associados, seja possível fazer a inserção de todo o conjunto de dados através de apenas um único pedido ao serviço.

```
curl -X POST \
  -d '{ "phone" : "{"phone_number": "+351 222 222 222"}", (...) }'\
  https://api.apey-eye.com/restaurant/
```

Listing 4.27: Inserção de dados com referências

```
curl -X POST \
  -d '{ (...),
        "addresses": [  { "address": "restaurantAddress1" },
                        { "address": "restaurantAddress2" } ] }'\
  https://api.apey-eye.com/restaurant
```

Listing 4.28: Inserção de dados com coleção

4.4.6 Documentação

Tal como foi desenhado na conceção da solução, também a implementação da *framework* teve em conta a integração de dois tipos de documentação que se encontrará disponível aos seus clientes.

HTTP OPTIONS

Um dos tipos de documentação da API é disponibilizado através do método HTTP OPTIONS. A partir da raiz da API ou do endereço de um determinado recurso é possível ter acesso não só à listagem dos recursos mas também, respetivamente, às diferentes propriedades que estão associadas à utilização do recurso em causa.

Na raiz da API a documentação apresentada não será nada mais do que a lista dos recursos associados acompanhados do endereço através do qual estarão disponíveis. Além disso, no caso de o serviço estar assente num *GenericRouter* é também apresentado o endereço através do qual é disponibilizada a interface *NoBackend*.

```
{
    "resources": {
        "restaurant": "http://localhost/classes/restaurant",
        "phone": "http://localhost/classes/phone",
        "category": "http://localhost/classes/category",
        "addresses": "http://localhost/classes/addresses",
        "categoryrestaurant": "http://localhost/classes/catrest"
    },
    "no_backend": "http://localhost/classes/{path}"
}
```

Listing 4.29: Exemplo de documentação da raiz da API

No que diz respeito aos recursos a documentação foca, para cada um dos métodos disponíveis, os parâmetros de consulta pré-definidos, bem como as propriedades de renderização e validação dos dados, os parâmetros de autenticação e autorização e ainda a estrutura de dados usada para a descrição da *Resource* ou de cada um dos métodos.

Além dos aspetos já referidos, a estrutura retornada aos clientes apresenta também os diferentes tipos de ações disponibilizados.

```
{
    "collection": {
        "output": {
            "_embedded": [ "phone" ]
        },
        "allowed_roles": [ "restaurant_owner", "admin" ],
        "auth": "basic",
        "documentation": {
            "title": "Restaurant Resource",
            "description": "Resource description."
        },
        "formatters": [ "application/json" ]
    },
    "methods": [
        {
            "http_method": "GET"
            "query" : {
                "_sort" : ["-name", "address"]
            }
        },
        {
            "http_method": "POST",
            "output": {
                "_embedded": [ "phone" ]
            },
            "allowed_roles": [ "restaurant_owner", "admin" ],
            "auth": "basic",
            "documentation": {
                "title": "Restaurant Resource",
                "description": "Resource description"
            }
        },
        {
            "http_method": "OPTIONS"
        }
    ],
    "actions": {
        "schema": {
            "http_method": "GET",
            "path": "/schema"
        }
        "count": {
            "http_method": "GET",
            "path": "/count"
        }
    }
}
```

Listing 4.30: Exemplo de documentação de um recurso

Documentação Interativa

Para a documentação interativa foi integrada na *framework* um módulo de Swagger[1], que permite tanto a disponibilização da documentação da API de uma forma visualmente intuitiva como o teste dos diferentes *endpoints*. No entanto, a inclusão deste módulo está dependente do tipo de *middleware* de reencaminhamento de pedidos utilizado dificultando assim a disponibilização de uma solução transversal a qualquer tipo de *Router* utilizado. Ciente desta dificuldade foi tomada a decisão de, nesta fase de desenvolvimento, apenas oferecer este tipo de documentação através do *Router* implementado através da *microframework* Hapi, dado que esta possui uma extensão que pode ser registada de modo a facilitar a disponibilização deste tipo de documentação.

Assim, quando utilizado um *HapiRouter* os utilizadores podem ter acesso a uma interface de documentação e testes semelhante na figura 4.3.

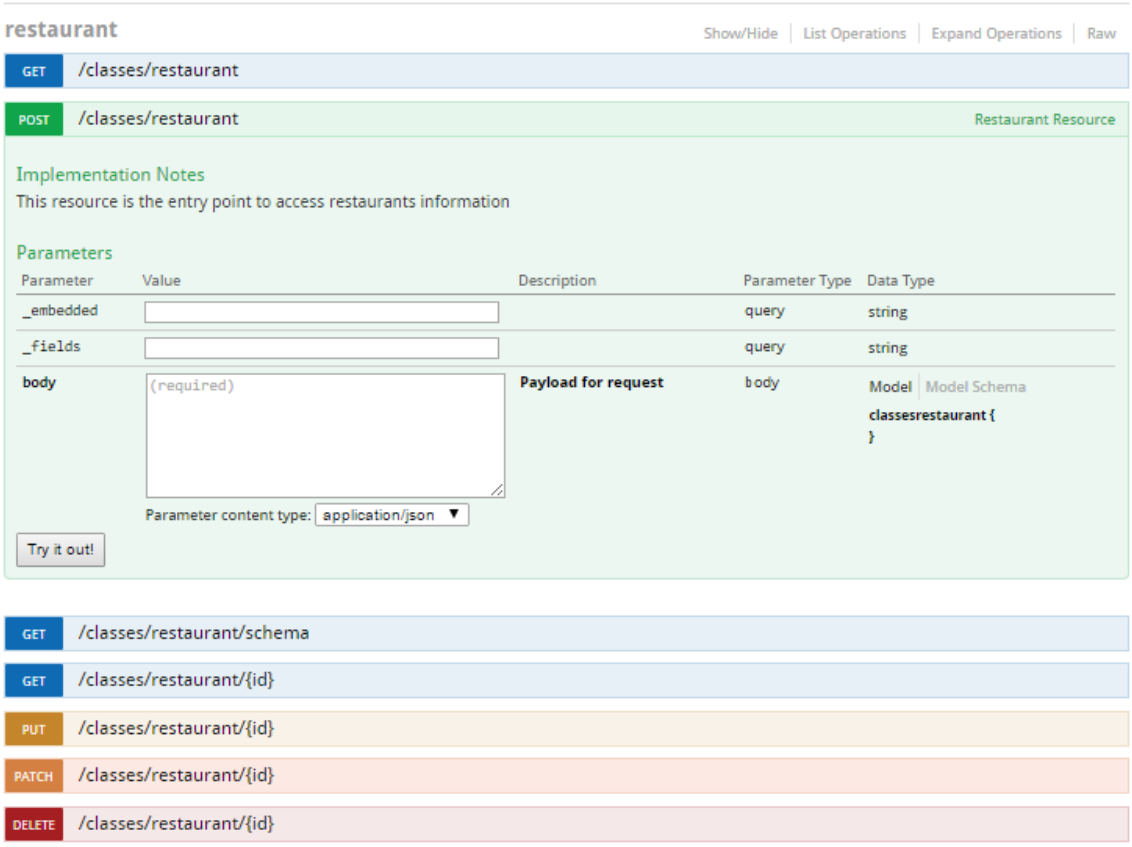

Figura 4.3: Documentação interativa através de Swagger

[1] http://swagger.io/

Capítulo 5

Validação

Este capítulo tem como principal objetivo apresentar a forma como a solução implementada foi validada. Para este processo foram utilizados diferentes mecanismos com foco na validação de diferentes aspetos, tais como a validação das funcionalidades da *framework*, a comparação do modo de utilização por parte dos programadores relativamente a outras soluções existentes e ainda através dos comentários obtidos da comunidade de utilizadores do tipo de tecnologia utilizada mas também de desenvolvimento de serviços REST.

5.1 Validação de Funcionalidades

Nesta dissertação, a validação das funcionalidades serviu, tal como o nome indica, para testar o correto funcionamento das diferentes funcionalidades da *framework* implementada.

Este processo ocorreu ao longo de toda a etapa de implementação e permitiu que, durante todas as fases, fosse possível verificar se a solução correspondia realmente à solução que tinha sido previamente desenhada.

5.1.1 Testes Unitários

O processo de desenvolvimento da solução planeada foi acompanhado também da implementação de testes unitários que permitiram ter o controlo do modo de funcionamento da *framework* através do teste do comportamento de cada uma das entidades envolvidas. Deste modo, para cada uma das entidades de maior impacto para a *framework*, foram desenvolvidos testes unitários, capazes de testar cada uma das suas funcionalidades de modo individual.

Apesar do objetivo dos testes unitários ser o teste individual das diferentes entidades de um sistema e de cada uma das suas funcionalidades, nesta *framework* apenas algumas das entidades foram alvo direto de testes unitários, tais como *Inputs*, *Models*, *Resources* e *Routers*, permitindo que ao longo da implementação fosse possível validar cada uma das suas funcionalidades e garantir o seu correto funcionamento da *framework*.

Apesar de indiretamente, as outras entidades da *framework* foram também testadas por intermédio do teste das anteriormente referidas. Dado que a solução desenvolvida envolve uma

ligação entre as várias entidades, à medida que uma determinada entidade é testada também aquelas que se relacionam são possíveis de ser testadas, tendo sido esta a abordagem utilizada para os testes unitários implementados.

No que diz respeito à conjugação entre os testes e o desenvolvimento, a abordagem utilizada com grande parte das funcionalidades da *framework* foi, em primeiro lugar, a implementação da funcionalidade em si sendo o desenvolvimento dos testes apenas efetuado após a funcionalidade se encontrar implementada. A opção de implementação de testes apenas após a implementação, apesar de não ser o aconselhado pelas práticas de desenvolvimento guiado por testes, tornou-se a mais viável tendo em conta a tipologia do projeto desenvolvido. Como a solução foi sofrendo evolução relativamente à sintaxe como seria oferecida aos programadores, a implementação de testes unitários envolveria um maior esforço de *refactoring* dos próprios testes.

Com isto, apesar de nem todas as entidades e funcionalidades terem sido diretamente testadas, o conjunto de testes unitários implementados permitiu obter uma cobertura de grande parte do código produzido, tendo sido possível obter valores de cobertura de código que rondam os 80%, tal como pode ser verificado no anexo D.

5.1.2 Prova de Conceito

Como forma de teste das diferentes funcionalidades da *framework* também se procedeu ao desenho e implementação de uma API que, embora simples, permitiu cobrir grande parte dos casos de uso da solução implementada.

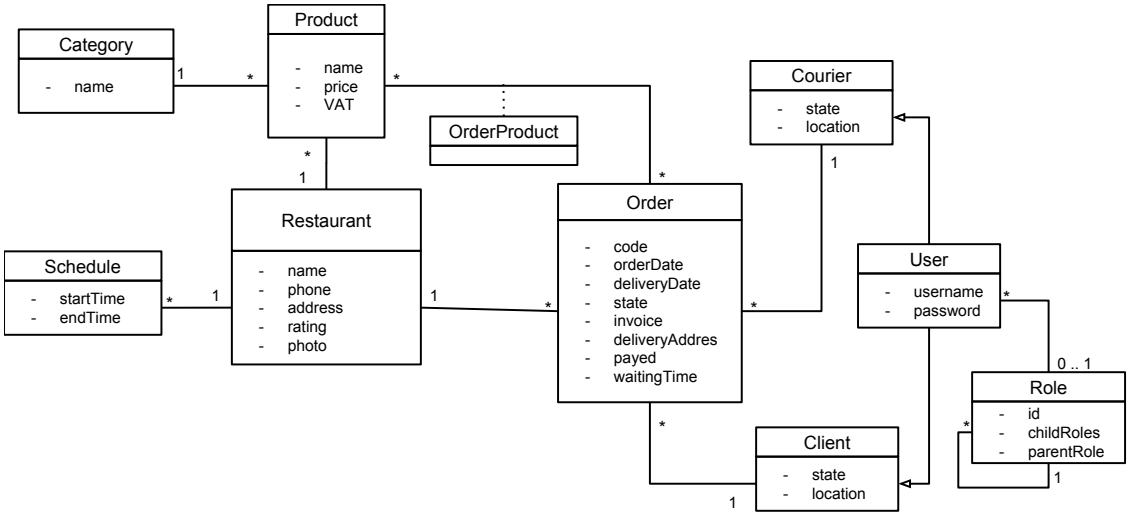

Figura 5.1: Esquema de dados da API implementada como prova de conceito

O serviço implementado pretende representar um excerto de um serviço de apoio a uma aplicação móvel de entrega de refeições, na qual diferentes entidades como restaurantes, produtos, categorias, encomendas e diferentes tipos de utilizadores interagem entre si. Foi adotada a implementação deste serviço pelo facto de se tratar de um caso real que se encontra em

desenvolvimento por outra dissertação no mesmo ambiente empresarial, e assim, permite ter uma maior perceção das necessidades reais de um serviço e a capacidade da solução desenvolvida satisfazer essas necessidades.

Para além da especificação de cada um dos modelos de dados enunciados, e ao mesmo tempo dos esquemas de validação de dados que lhes são associados, a construção do serviço passou também pela implementação da lógica de negócio da aplicação através do desenvolvimento de cada uma das *Resources* que serão disponibilizadas.

Com isto foi possível introduzir regras de negócio na API de uma forma bastante simples, implementar as diferentes relações entre os recursos, limitar o acesso aos objetos introduzindo a necessidade de autenticação dos utilizadores e ainda limitar o tipo de operações que cada um dos grupos de utilizadores possuíam permissões de utilizar.

Além disso foi possível implementar ações personalizadas nas *Resources* de uma forma relativamente simples, substituir a implementação base já existente no *GenericResource* ou implementar uma *Resource* personalizada sem que o programador fosse obrigado a um esforço de implementação considerável.

Assim, através do desenvolvimento desta prova de conceito, embora represente um serviço relativamente simples, foi possível perceber que a solução implementada permite que seja construído um serviço de uma forma relativamente rápida e simples, abstraindo tarefas que apesar de simples fazem com que o desenvolvimento do serviço envolva um esforço de implementação maior, tais como o acesso à base de dados, o controlo de acesso, a associação dos recursos aos *routers* e ainda o tratamento dos diferentes tipos de propriedades passíveis de ser recebidas dos clientes. Além disso foi também possível disponibilizar uma documentação da API de uma forma completamente transparente para o programador do serviço, promovendo uma descoberta mais facilitada dos seus métodos bem como uma melhor compreensão do serviço por parte dos seus clientes.

5.2 Comparação com soluções existentes

Como forma de validação foi também feita uma comparação entre a forma de utilização de algumas das *frameworks* já existentes para a implementação de serviços REST para Node.js, como Koa e Hapi, e a solução implementada através desta dissertação.

5.2.1 Mapeamento pedidos HTTP

Como ponto inicial foi feita a implementação de um serviço muito simples através das três *frameworks*, na qual o objetivo seria a devolução de uma resposta bastante simples para cada um dos métodos HTTP, comparando assim a forma como ocorre o mapeamento entre os diferentes tipos de pedidos HTTP e o seu tratamento.

Koa

Como é possível verificar pelo anexo E.1, através desta *framework* é necessário que o programador, para além do Koa, também utilize uma entidade *router* por forma a implementar o *middleware* para cada uma das rotas. Além disso obriga a que o programador tenha uma intervenção direta a um nível mais baixo do tratamento de cada um dos pedidos recebidos, tendo a necessidade de indicar, para cada um deles, explicitamente o tipo do pedido e ainda a rota no qual este irá ser disponibilizado. Através desta solução o *handler* do pedido é ainda dependente da *framework*, obrigando a que neste caso seja uma função geradora.

```
1  var koa = require('koa'),
2     route = require('koa-route'),
3     app = koa();
4
5 app.use(route.get('/api/items/:id', function*(id) {
6     this.body = 'Get id: ' + id;
7 }));
8
9 var server = app.listen(3000, function() {
10   console.log('Koa is listening to http://localhost:3000');
11 });
```

Listing 5.1: Excerto da implementação de um serviço REST através de Koa

Hapi

Apesar de a implementação através de Hapi também não evitar a elevada repetição de código necessário para a definição das diferentes rotas do serviço, como pode ser verificado no anexo E.2, ao contrário do que acontece com o Koa, a definição das rotas é apresentada ao programador de uma forma mais configurável. No entanto obriga igualmente a preocupações de mais baixo nível de reencaminhamento de pedidos para cada um dos métodos disponibilizados tais como a rota na qual serão disponibilizados.

Dado ser mas configurável faz com que a sua utilização seja visualmente mais limpa e mais facilmente interpretável comparando com outras *frameworks* como o Koa ou até o Express.

```
1 var Hapi = require('hapi');
2 var server = new Hapi.Server(3000);
3
4 server.route([{
5         method: 'GET',
6         path: '/api/items/{id}',
7         handler: function(request, reply) {
8           reply('Get item id: ' + request.params.id);
9         }
```

```
    }]
 );

server.start(function() {
  console.log('Hapi is listening to http://localhost:3000');
});
```

Listing 5.2: Excerto da implementação de um serviço REST através de Hapi

Solução implementada

Relativamente à construção de um serviço REST, embora muito simples e minimalista, tal como pode ser verificado no anexo E.3, a solução implementada por esta dissertação permite, em primeiro lugar, facilitar o registo de métodos no *middleware* de reencaminhamento de pedidos HTTP. Pelo facto de ser baseado em classes, a *framework* desenvolvida permite que todos os métodos pertencentes a um único recurso da API possam ser todos englobados todos na mesma entidade que será posteriormente registada no *Router*, sendo responsabilidade interna à *framework* a associação de cada uma dos métodos da entidade aos métodos e rotas correspondentes.

Relativamente à associação das *Resources* ao *Router* , desde que tenha sido atribuído um identificador à *Resource*, o programador apenas necessita de registar cada uma das *Resources* sem se preocupar com o mapeamento de cada um dos seus métodos, ao contrário do que acontece com as *frameworks* referidas anteriormente, sendo os métodos automaticamente mapeados para o seu tipo de pedido HTTP correspondente.

```
import ApeyEye from '../apey-eye';

let HapiRouter = ApeyEye.HapiRouter,
    HapiGenericRouter = ApeyEye.HapiGenericRouter,
    Name = ApeyEye.Annotations.Name;

@Name('items')
class ItemResource extends ApeyEye.Resource{
    static async fetchOne(options){
        return "GET id:" + options.id;
    }
}

let router = new HapiGenericRouter();
router.register([{ resource: ItemResource }]);

router.start({port: 3000}, function (err, server) {
    console.log('Server running at', server.info.uri);
});
```

Listing 5.3: Excerto da implementação de um serviço REST através da solução implementada

5.2.2 Acesso à base de dados

Também no que diz respeito ao acesso à base de dados, a solução implementada pretende ser facilitadora, relativamente às outras soluções em estudo.

Koa e Hapi

Tanto para a *framework* Koa como para a Hapi, a forma de acesso à base de dados deve ser definida diretamente no corpo do método responsável. Apesar de ser igualmente possível fazer a utilização de um ORM que permita abstrair o acesso à base de dados, este deve ser igualmente utilizado pelo programador do serviço nos respetivos métodos.

```
app.get('api/items/', function* get(next) {
    try{
        var cursor = yield r.table('items').run(this.databaseConnection);
        var result = yield cursor.toArray();
        this.body = JSON.stringify(result);
    }
    catch(e) {
        this.status = 500;
        this.body = e.message || http.STATUS_CODES[this.status];
    }
    yield next;
});
```

Listing 5.4: Acesso à base de dados através de Koa

```
server.route([{
        method: 'GET',
        path: '/api/items/{id}',
        handler: function(request, reply) {
          var r = this.rethinkdb;
          var conn = this.rethinkdbConn;

          r.table('items').getAll().run(conn, function (err, cursor){
            reply(cursor.toArray());
          });
        }
    }]
 );
```

Listing 5.5: Acesso à base de dados através de Hapi

Solução implementada

Através da solução implementada, para o acesso a uma determinada tabela da base de dados apenas é necessário fazer a criação de um *Model* que faça o mapeamento dos objetos da base de dados para JavaScript.

Após a criação do *Model*, este pode ser diretamente usado na implementação de cada um dos métodos, tal como acontece com outros tipos de ORMs, com a vantagem de ter uma interface semelhante às *Resources*, ou ainda ser associado a uma *GenericResource* para que seja automaticamente utilizado sem a intervenção do programador.

```
1 @Name('items')
2 class ItemsModel extends ApeyEye.RethinkDBModel{}
3
4 class ExampleResource extends ApeyEye.Resource{
5     static async fetch(options){
6         let modelObj = await ModelClass.fetch({resourceProperties:options.
              requestProperties});
7         return ResourceClass._serializeArray(modelObj, properties);
8     }
9 }
```

Listing 5.6: Acesso à base de dados através da solução implementada

5.2.3 Autenticação

Koa

Para a *framework* Koa, a forma de verificar a autenticação é obrigar a que seja adicionado um *middleware* ao tratamento do pedido no qual este é verificada a autenticação. Além deste pormenor é ainda necessário que o programador associe ao serviço as extensões necessárias para que os mecanismos de autenticação sejam incluídos.

```
1  function *authed(next){
2    if (this.req.isAuthenticated()){
3      yield next;
4    } else {
5      this.body = 401;
6    }
7  }
8  app.get('api/items/',authed, function* get(next) {
9      yield next;
10 });
```

Listing 5.7: Verificar autenticação através de Koa

Hapi

No que diz respeito à *framework* Hapi, é relativamente simples de indicar que um determinado pedidos deve passar por um processo de autenticação, sendo apenas necessário mencionar o identificador do esquema de autenticação a ser usado. Além disso, tal como acontece com o Koa, é igualmente necessário, antes de inicializar o serviço, incluir no mesmo as extensões responsáveis pelo tratamento dos mecanismos de autenticação.

```
1  server.route({
2          method: 'GET',
3          path: '/',
4          config: {
5              auth: 'basic',
6              handler: function (request, reply) {
7                  reply('hello, ' + request.auth.credentials.name);
8              }
9          }
10     });
```

Listing 5.8: Verificar autenticação através de Hapi

Solução implementada

Através da solução implementada, a indicação da necessidade de autenticação para um determinado método, ou até para todos os métodos de uma classe *Resource* pode ser feita simplesmente através da utilização do decorador *@Authentication*.

Indicar o método de autenticação a ser utilizado é o único pormenor com que o programador se deve preocupar, todo o restante processo de autenticação é efetuado automaticamente pela *framework*.

```
1  class ExampleResource extends ApeyEye.Resource{
2      @Authentication('basic')
3      static async fetch(options){     }
4  }
```

Listing 5.9: Verificar autenticação através da solução implementada

5.2.4 Autorização

Koa

De modo semelhante à autenticação, também para a autorização a *framework* Koa, obriga a que o programador adicione um *middleware* ao pedido com o objetivo de verificar o papel de cada utilizador no sistema.

```
function *authorized(next){
  if (this.user.role === "ADMIN"){
    yield next;
  } else {
    this.body = 403;
  }
}
app.get('api/items/',authorized, function* get(next) {
    yield next;
});
```

Listing 5.10: Autorização através de Koa

Hapi

Também para a autorização o Hapi possui extensões capazes de lidar com a gestão de acessos relacionado com os grupos de permissões que cada um dos utilizadores possui. Neste sentido é apenas necessário incluir uma extensão para esse objetivo e em cada uma das rotas indicar os grupos de utilizadores autorizados.

```
server.route({
    method: 'GET',
    path: '/',
    config: {
        plugins: {'hapiAuthorization': {role: 'ADMIN'}},
        handler: function (request, reply) { reply("Great!");}
}});
```

Listing 5.11: Autorização através de Hapi

Solução Implementada

Através da solução implementada, é possível utilizar decoradores específicos para indicar quais os grupos de utilizadores autorizados e todo o processo de verificação é tratado internamente pela *framework*.

```
class ExampleResource extends ApeyEye.Resource{
    @Authentication('ADMIN')
    static async fetch(options){    }
}
```

Listing 5.12: Autorização através da solução implementada

5.3 Feedback da comunidade

Como forma de validação da solução implementada foi também compilado todo o trabalho produzido num único módulo de JavaScript, o qual foi disponibilizado para a comunidade de desenvolvimento *Open Source* através de um repositório público no GitHub e um registo de módulos utilizado para esse efeito, o *npm Registry*.

Um dos objetivos iniciais desta dissertação era a disponibilização de uma solução *Open Source* e através deste processo foi possível publicar uma solução com a comunidade por forma a que fosse possível ser usada e testada por programadores da área. Além disso, foi associada ao projeto uma licença *open source* criada pelo *Massachusetts Institute of Technology* (MIT) que permite a utilização da solução de qualquer forma e com qualquer tipo de *software*, livre ou não.

Assim, como forma de validação da solução implementada, para além de ter sido disponibilizada a *framework* para a comunidade, esta foi divulgada em alguns fóruns e blogues da área por forma a que fosse possível receber *feedback* de um conjunto maior de utilizadores. Além disso, o trabalho desenvolvido também foi apresentado num encontro de entusiastas de novas tecnologias *web*, no qual foi possível ter contacto com programadores de diversificadas áreas e debater sobre a solução implementada.

Através da divulgação a vários níveis foi possível perceber que apesar de o desenho de uma solução que usa como base um mapeamento objeto-recurso não ser algo completamente novo na comunidade de desenvolvimento de serviços REST, é uma das características praticamente inexistente nas soluções já existentes para utilização através de Node.js e de que os programadores mais sentem falta quando comparando com *frameworks* como Rais ou Django REST Framework. Segundo alguns comentários obtidos foi possível perceber que este foi um dos aspetos que mais impacto causou na solução implementada e que a tornaram algo inovador e vantajoso em relação às soluções já existentes. Uma das vantagem enunciadas pela comunidade foi o facto de, através dos diferentes tipos de mapeamentos proporcionados pela *framework* ser possível que o tempo gasto com o desenvolvimento dos serviços seja aproveitado para a implementação da lógica de cada um dos recursos disponibilizados ao invés de ser gasto com questões de servidores HTTP e de acesso à camada de base de dados, que à partida são questões de mais baixo nível que os programadores têm de suportar em todos os serviços desenvolvidos, e que assim podem ser abstraídas.

Relativamente à forma como o JavaScript foi utilizado, nomeadamente à utilização das características mais recentes, este é um dos aspetos que deixa a comunidade mais relutante quanto à utilização da *framework*, dado estas mudarem a perspetiva de utilização da linguagem bem como obrigarem a uma transformação do código produzido. Apesar disso, a legibilidade e a elegância introduzida por estas caraterísticas na forma de programar em JavaScript fazem com que a solução seja de certo modo atrativa e faz com que alguns programadores acreditem que a *framework* é uma solução útil para ser utilizada como forma de os programadores começarem a aprender e a utilizar as novas caraterísticas do JavaScript o mais cedo possível, mesmo antes de estas estarem implementadas de forma nativa.

Capítulo 6

Conclusões

Após todos os capítulos anteriores relativos tanto à fase de análise do estado da arte mas também relativos às fases de conceção, implementação e validação da solução, este último capítulo pretende, além de fazer uma análise do impacto das diferentes escolhas tecnológicas tomadas, fazer um breve resumo e avaliação do trabalho produzido. Neste capítulo são ainda apresentadas as ambições para a continuidade do trabalho desta dissertação sendo delineadas algumas perspetivas pelas quais seria interessante haver uma evolução do trabalho produzido.

6.1 Avaliação das escolhas tecnológicas

A solução desenvolvida, apresentada como uma *framework* em Node.js para o desenvolvimento de serviços REST, utiliza as características de JavaScript de próxima geração, ou seja, características que no decorrer desta dissertação ainda representavam uma versão de JavaScript, que não se encontra implementada de forma nativa. Algumas destas características, ainda em fase de implementação no JavaScript ou até em fase de proposta de especificação, permitiram dotar a solução de características raras que a distinguem de outras soluções já existentes no mercado.

Atualmente a utilização deste tipo de tecnologias implica algum esforço inicial no desenvolvimento de aplicações, sendo necessário, em primeiro lugar, compreender a forma de funcionamento destas novas caraterísticas, mas também a associação de algum tipo de transpilador de código que seja capaz de transformar o código produzido para uma sintaxe capaz de ser executada pelo JavaScript.

A necessidade de transformação do código fonte pode ser considerado um entrave à adoção de *frameworks* baseadas neste tipo de caraterísticas no entanto o desenvolvimento de soluções prevendo este tipo de evoluções tecnológicas permite, de certo modo, antecipar e promover a utilização das mesmas. Assim, aquando da sua integração de modo nativo, soluções como a que foi desenvolvida nesta dissertação podem ser o ponto de partida para uma adoção mais facilitada das tecnologias.

Tal como grande parte das evoluções tecnológicas, apesar deste tipo de funcionalidades da linguagem poderem levar a um atraso no arranque do desenvolvimento de um determinado projeto, após a fase inicial podem em muito contribuir para um desenvolvimento mais rápido e acessível aos programadores. Este foi uma das conclusões alcançadas através da concretização desta dissertação, na qual a adoção das novas caraterísticas do JavaScript deram a capacidade de desenvolver uma solução de uma forma mais simples, mais facilmente compreensível ao programador e ao mesmo tempo oferecendo soluções que de outro modo exigiriam um esforço de implementação maior.

6.2 Avaliação do trabalho desenvolvido

O trabalho desenvolvido durante esta dissertação permitiu que fosse possível apresentar uma solução bastante completa para uma *framework* orientada ao desenvolvimento de serviços REST.

A *framework* implementada permite que os programadores consigam construir as suas APIs de uma forma bastante simples e modular, através de uma solução onde as diferentes responsabilidades são claramente delimitadas pelas diferentes entidades envolvidas. Esta clara divisão permite que, ao longo de todo o fluxo da aplicação, haja um mapeamento objeto-recurso, no qual os objetos passíveis de ser comunicados aos clientes possuem uma representação orientada a objetos desde a sua obtenção da base de dados até à sua comunicação aos clientes.

Esta foi uma das principais vantagens alcançadas com esta solução, permitindo que fosse possível manipular e utilizar as diferentes entidades de um modo independente dos pedidos HTTP, proporcionando uma interface programática possível de ser utilizada tanto aliada a *routers* como de um modo isolado, facilitando, entre outras coisas, o teste das mesmas.

Além disso, um outro objetivo alcançado, graças à modularidade da solução obtida, foi a capacidade de oferecer aos utilizadores da *framework* alguma liberdade na escolha das tecnologias que lhes servirão de apoio à implementação dos serviços. Em primeiro lugar a solução implementada possui independência total do tipo de base de dados utilizado, ou seja, é permitido ao programador construir o seu próprio módulo, sendo apenas obrigado à implementação dos métodos específicos de contacto com o sistema de base de dados, dado que toda a lógica de negócio relativa aos *Models* se encontra isolada num outro componente.

A *framework* permite ainda que o programador escolha o tipo de *middleware* HTTP que pretende utilizar, podendo, tal como acontece com os *Models*, implementar a sua lógica específica ao *middleware* pretendido. Apesar da possibilidade, esta encontra-se de certo modo limitada e menos independente do restante sistema do que o que acontece com os *Models*. Funcionalidades tais como a disponibilização de uma API genérica ou ainda a disponibilização de uma documentação automática através de *Swagger* encontram-se dependentes da *micro-framework* utilizada para o tratamento dos pedidos HTTP e, por isso, apesar de os utilizadores terem a liberdade de implementarem o seu próprio *Router*, é aconselhável que os programadores utilizem apenas as entidades oferecidas pela solução implementada ou, por outro

lado, tenham em consideração a lógica de negócio da solução implementada de modo a implementarem o seu próprio *Router* corretamente e atendendo aos requisitos da *framework*.

Um dos problemas que esta dissertação teve sempre em atenção foi a capacidade de os serviços construídos disponibilizarem, de um modo automático, uma interface com documentação relativa aos diferentes métodos disponíveis. Apesar das limitações indicadas anteriormente, relacionadas com os *Routers*, a solução possibilita que os serviços desenvolvidos ofereçam aos seus clientes não uma mas duas formas de documentação automática, adaptadas aos diferentes tipos de clientes das APIs.

Apesar de não terem sido implementados, a solução contemplou um outro conjunto de módulos que de certa forma podem ser também associados à documentação dos serviços. Um dos objetivos da disponibilização da documentação é proporcionar uma forma de os clientes perceberem o modo como irão interagir com os serviços e é neste sentido que o desenho de uma solução para a componente cliente foi considerada essencial. Deste modo, uma das melhores formas encontradas para facilitar o acesso aos serviços é oferecer aos clientes *Models* para a componente cliente que lhes permitam aceder aos mesmos de um modo abstraído de determinadas preocupações aliadas à comunicação com entidades remotas. Este pormenor de desenho da solução além de permitir diminuir o grau de complexidade de acesso aos serviços construídos permitiu ainda que outras funcionalidades fossem projetadas, tais como a abordagem reativa na transmissão de dados e a *cache* do lado do cliente, o que permitirá que a *framework* seja usada em abordagens mais recentes de desenvolvimento de aplicações.

Por último, como é possível comprovar por todo o trabalho desenvolvido nesta dissertação, o cumprimento das restrições definidas pela arquitetura REST foi uma preocupação constante e através da qual foi possível desenhar uma solução completa ao nível de arquitetura. A *framework* desenvolvida faz uma distinção clara de quais as componentes cliente e servidor. Além dessa restrição, a interface uniforme e a elaboração de um sistema em camadas foram as restrições que mereceram uma maior atenção e um grande foco de implementação. Além disso, tanto a *cache* como a possibilidade de disponibilização de código de cliente foram restrições que, embora não implementadas também fizeram parte da solução implementada. Por último, a filosofia da *framework* também assenta numa perspetiva perspetiva na qual o servidor não armazena dados entre diferentes pedidos, à exceção do armazenamento persistente das informações relativas aos recursos da API.

Embora a temática em estudo por esta dissertação não possa ser considerado algo completamente inovador, por já existir um vasto conjunto de soluções no mercado para o desenvolvimento de serviços REST através das mais variadas linguagens, a forma como a mesma foi organizada e as tecnologias utilizadas na sua implementação podem ser considerados o principal foco de inovação. Deste modo, a inovação não se centra no produto no seu contexto principal, mas na forma como o mesmo foi implementado e como o mesmo se apresenta aos seus utilizadores, através de um conjunto de funcionalidades atualmente menos comuns para este tipo de soluções.

6.3 Trabalho Futuro

Atendendo à componente da solução desenhada que não foi implementada, esta dissertação deixa em aberto o desenvolvimento de um conjunto de funcionalidades que em muito permitirão enriquecer a solução desenvolvida.

Uma das preocupações que deve ser atendida é a inclusão de mecanismos que permitem aumentar a performance do serviço, através da integração camadas de *cache* que, tal como foi apresentado nesta dissertação, poderá diminuir a carga de trabalho e consecutivamente o tempo de resposta para o tratamento dos pedidos recebidos.

A capacidade de integração da *framework* na componente cliente dos sistemas seria também um dos focos a desenvolver no futuro. A utilização de *Models* nas aplicações cliente, além de abstrair as tarefas relativas ao acesso a serviços remotos, permitiria também criar uma unificação das interfaces utilizadas nas diferentes realidades do serviço. Além disso, a utilização deste tipo de entidades permitiria, que de um modo transparente, fossem incluídas outro tipo de caraterísticas na *framework* que permitiriam a sua utilização em aplicações com abordagens de transmissão de dados em tempo real ou ainda o desenvolvimento de aplicações orientadas ao funcionamento *offline*. Estas caraterísticas, apesar de ainda relativamente raras, permitiriam melhorar consideravelmente a experiência de utilização das aplicações desenvolvidas.

Aliado à separação das responsabilidades de um serviço, à separação da carga de trabalho e mais uma vez à performance geral de um serviço, um dos pontos mais ambiciosos para o trabalho futuro é evolução a *framework* de modo a permitir não só o desenvolvimento de serviços REST convencionais mas também o desenvolvimento de sistemas baseados numa arquitetura de micro serviços. Este seria um dos pontos essenciais para permitir a sua utilização em projetos onde a escalabilidade é uma preocupação inerente, permitindo assim uma separação real das diferentes responsabilidades em serviços independentes.

Referências

[Abe11] Michael Abernethy. *Just what is Node.js*. IBM Corporation, 2011. URL: http://pt.scribd.com/doc/54504131/What-is-node-JS-developerWorks#scribd.

[Arc14] Jake Archibald. JavaScript Promises, 2014. URL: http://www.html5rocks.com/en/tutorials/es6/promises/.

[Avr14] Abel Avram. The Strengths and Weaknesses of Microservices, 2014. URL: http://www.infoq.com/news/2014/05/microservices.

[BHM+04] David Booth, Hugo Haas, Francis McCabe, Eric Newcomer, Michael Champion, Chris Ferris e David Orchard. Web Services Architecture, 2004. URL: http://www.w3.org/TR/ws-arch/.

[BN84] Andrew D. Birrell e Bruce Jay Nelson. Implementing remote procedure calls. *ACM Transactions on Computer Systems*, 2(1):39–59, February 1984. URL: http://portal.acm.org/citation.cfm?doid=2080.357392, doi:10.1145/2080.357392.

[CC13] Kim-Mai Cutler e Josh Constine. Facebook Buys Parse To Offer Mobile Development Tools As Its First Paid B2B Service, 2013. URL: http://techcrunch.com/2013/04/25/facebook-parse/.

[CKT14] Ioannis K. Chaniotis, Kyriakos-Ioannis D. Kyriakou e Nikolaos D. Tselikas. Is Node.js a viable option for building modern web applications? A performance evaluation study. *Computing*, March 2014. URL: http://link.springer.com/10.1007/s00607-014-0394-9, doi:10.1007/s00607-014-0394-9.

[Cog13] Arnaud Cogoluègnes. Documenting a REST API with Swagger and Spring MVC, 2013. URL: http://blog.zenika.com/index.php?post/2013/07/11/Documenting-a-REST-API-with-Swagger-and-Spring-MVC.

[CPDM14] Bruno Costa, Paulo F. Pires, Flavia C. Delicato e Paulo Merson. Evaluating a Representational State Transfer (REST) Architecture: What is the Impact of REST in My Architecture? *2014 IEEE/IFIP Conference on Software Architecture*, pages 105–114, April 2014. URL: http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=6827107, doi:10.1109/WICSA.2014.29.

[Cro] Douglas Crockford. Classical Inheritance in JavaScript. URL: http://www.crockford.com/javascript/inheritance.html.

[Dav07] DavidT. The evolution of Web Services, 2007. URL: http://www.techmachina.com/2007/08/evolution-of-web-services.html.

[Elk08a] M. Elkstein. REST as Lightweight Web Services, 2008. URL: http://rest.elkstein.org/2008/02/rest-as-lightweight-web-services.html.

[Elk08b] M. Elkstein. What is REST?, 2008. URL: http://rest.elkstein.org/2008/02/what-is-rest.html.

[Fie00] RT Fielding. Architectural styles and the design of network-based software architectures. 2000. URL: http://jpkc.fudan.edu.cn/picture/article/216/35/4b/22598d594e3d93239700ce79bce1/7ed3ec2a-03c2-49cb-8bf8-5a90ea42f523.pdf.

[Fre13] Todd Fredrich. *RESTful Service Best Practices Recommendations for Creating Web Services*. 2013.

[Gom08] Guilherme Gomes. *Tábula: uma framework para o desenvolvimento de aplicações REST*. PhD thesis, Universidade da Madeira, 2008. URL: http://digituma.uma.pt/handle/10400.13/110.

[Hem13] Zef Hemel. NoBackend: Front-End First Web Development, 2013. URL: http://www.infoq.com/news/2013/05/nobackend.

[HK97] Markus Horstmann e Mary Kirtland. DCOM architecture. *Microsoft Corporation, July*, 1997. URL: http://softwaretoolbox.com/dcom/DCOMArchitecture.pdf.

[Hof14] Todd Hoff. Microservices - Not A Free Lunch!, 2014. URL: http://highscalability.com/blog/2014/4/8/microservices-not-a-free-lunch.html.

[Hol14] Alex Holmes. Reviewing Django REST Framework, 2014. URL: http://blog.isotoma.com/2014/03/reviewing-django-rest-framework/.

[Hor12] Ryan Horn. Introducing Flask-RESTful, 2012. URL: https://www.twilio.com/engineering/2012/10/18/open-sourcing-flask-restful.

[Jam12] Justin James. 10 things you should do to write effective RESTful Web services, 2012. URL: http://www.techrepublic.com/blog/10-things/10-things-you-should-do-to-write-effective-restful-web-services/.

[Jon12] Darren Jones. Rails or Sinatra: The Best of Both Worlds? *sitepoint*, 2012. URL: http://www.sitepoint.com/rails-or-sinatra-the-best-of-both-worlds/.

[Jun12] FAR Junior. Programação Orientada a Eventos no lado do servidor utilizando Node.js. 2012. URL: http://www.infobrasil.inf.br/userfiles/16-S3-3-97136-Programa%C3%A7%C3%A3oOrientada___.pdf.

[KBM13] Kennedy Kambona, EG Boix e Wolfgang De Meuter. An evaluation of reactive programming and promises for structuring collaborative web applications. *Proceedings of the 7th Workshop on Dynamic Languages and Applications*, 2013. URL: http://dl.acm.org/citation.cfm?id=2489802.

[Laa13] Joahn Laanstra. *Offline Data and Synchronization for a Mobile Backend as a Service system.* PhD thesis, Delft University of Tecnology, 2013.

[LC11] Li Li e Wu Chou. Design and Describe REST API without Violating REST: A Petri Net Based Approach. *2011 IEEE International Conference on Web Services*, pages 508–515, July 2011. URL: http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=6009431, doi:10.1109/ICWS.2011.54.

[Leo] Marcus Rommel Barbosa Leopoldo. Simple Object Access Protocol, Entendendo o Simple Object Access Protocol (SOAP).

[Mat14] Andy Matthews. Beginner's Guide to the Django Rest Framework, 2014. URL: http://code.tutsplus.com/tutorials/beginners-guide-to-the-django-rest-framework--cms-19786.

[Net14a] Mozilla Developer Network. Iterators and Generators, 2014. URL: https://developer.mozilla.org/en-US/docs/Web/JavaScript/Guide/Iterators_and_Generators.

[Net14b] Mozilla Developer Network. Promise, 2014. URL: https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Promise.

[NSs14] Dmitry Namiot e Manfred Sneps-sneppe. On Micro-services Architecture. 2(9):24–27, 2014.

[PL08] Cesare Pautasso e Frank Leymann. RESTful Web Services vs . " Big " Web Services : Making the Right Architectural Decision Categories and Subject Descriptors. pages 805–814, 2008.

[Rev] Reverb. Swagger. URL: https://helloreverb.com/developers/swagger.

[Ric14] Chris Richardson. Microservices: Decomposing Applications for Deployability and Scalability, 2014. URL: http://www.infoq.com/articles/microservices-intro.

[RLBV10] Gregor Richards, Sylvain Lebresne, Brian Burg e Jan Vitek. An analysis of the dynamic behavior of JavaScript programs. *Proceedings of the 2010 ACM SIGPLAN conference on Programming language design and implementation - PLDI '10*, page 1, 2010. URL: http://portal.acm.org/citation.cfm?doid=1806596.1806598, doi:10.1145/1806596.1806598.

[Ros13] Mike Roslog. REST e SOAP: Usar um dos dois ou ambos?, 2013. URL: http://www.infoq.com/br/articles/rest-soap-when-to-use-each.

[Rou06] Margaret Rouse. CORBA (Common Object Request Broker Architecture), 2006. URL: http://searchsqlserver.techtarget.com/definition/CORBA.

[Sch14] Ricardo Schroeder. Arquitetura e Testes de Serviços Web de Alto Desempenho com Node.js e MongoDB. 2014.

[SE12] Charles Severance e Brendan Eich. JavaScript : a Language in 10 Days. *Computing Conversations*, (February):7–8, 2012.

[Sla14] Noah Slater. Backend as a Service, 2014. URL: https://blog.engineyard.com/2014/backend-as-a-service.

[TV10] Stefan Tilkov e S Vinoski. Node. js: Using JavaScript to build high-performance network programs. *IEEE Internet Computing*, 2010. URL: http://doi.ieeecomputersociety.org/10.1109/MIC.2010.145.

[W3C12] W3C. A Short History of JavaScript, 2012. URL: https://www.w3.org/community/webed/wiki/A_Short_History_of_JavaScript.

[Zak09] NC Zakas. Professional javascript for web developers. 2009. URL: http://read.uberflip.com/i/113144/44http://scholar.google.com/scholar?hl=en&btnG=Search&q=intitle:Professional+JavaScript+for+Web+Developers#0.

Anexo A

Comparação SOAP vs REST

As arquiteturas SOAP e REST são a seguir distinguidas através de uma análise comparativa, tendo em conta aspetos como o formato das mensagens, protocolo de transporte, design, descrição do serviço e segurança [Gom08].

Rede

Para as arquiteturas REST, a *Web* é considerado um meio de publicação do serviço, onde os recursos e as diferentes operações sobre os mesmos estão disponíveis através de URLs. Por outro lado, na perspetiva das arquiteturas SOAP, a *Web*, é entendida como uma forma de transporte de mensagens. Assim, "as aplicações ganham a habilidade de interagir remotamente através da *Web*, mas continuando "fora"da *Web*" [PL08].

Protocolo de Transporte

Optando pela implementação de um serviço REST, o único protocolo de transporte que pode ser considerado é o protocolo HTTP. Relativamente aos serviços SOAP, estes são independentes do protocolo de transporte utilizado, podendo haver troca de mensagens através de HTTP, TCP, SMTP, etc.

Formato das mensagens

Num serviço SOAP, as mensagens têm obrigação de obedecer ao formato *standard* estabelecido pela arquitetura, ou seja, XML.

No que diz respeito aos serviços REST, estes têm possibilidade de definir o formato das mensagens a trocar, sendo normalmente utilizados os formatos JSON ou XML. Este fator pode constituir uma desvantagem, podendo complicar a interoperabilidade do serviço, mas por outro lado, pode ser vantajoso, oferecendo um maior leque de formatos de representação dos recursos.

Descrição do serviço

Os serviços SOAP, utilizam e dependem de WSDL, um *standard* de descrição de serviços *Web*, capaz de descrever um serviço de um modo compreensível por outros componentes do sistema.

Relativamente aos serviços REST, a descrição dos mesmos é feita de um modo mais "humano", não havendo uma formalização destinada à descrição de serviços. No entanto, de modo a contornar esta desvantagem, existem já métodos de descrição de APIs, como por exemplo, Swagger, API Blueprint ou RAML, que pretendem dotar os serviços REST de uma forma de descrição *standard*.

Design

Enquanto numa arquitetura SOAP o serviço é orientado às ações, numa arquitetura REST o serviço é orientado aos recursos, no qual todas as ações levarão à manipulação direta de recursos.

Segurança

Em termos de segurança, tanto a arquitetura REST como a arquitetura SOAP pode ser implementada utilizando o protocolo HTTP sobre *Secure Sockets Layer* (SSL) para proporcionar comunicações seguras.

A arquitetura SOAP contém ainda um conjunto de opções que podem ser aplicadas à troca de mensagens entre os diferentes módulos de um sistema, permitindo assim que o serviço seja dotado de um nível de qualidade e segurança na comunicação de uma forma independente do protocolo de transporte utilizado.

Anexo B

Métodos e códigos de estado HTTP

B.1 Métodos

Método	**Contexto**	**Significado**
GET	Coleção	Retorna todos os recursos de uma coleção
GET	Recurso	Retorna um único recurso
HEAD	Recurso	Retorna todos os recursos de uma coleção (apenas o cabeçalho)
HEAD	Coleção	Retorna apenas um único recurso (apenas o cabeçalho)
POST	Cria um novo recurso	
PUT	Recurso	Atualiza um recurso
PATCH	Recurso	Atualiza um recurso
DELETE	Recurso	Elimina um recurso
OPTIONS	Qualquer	Retorna todos os métodos HTTP disponíveis e outras opções

B.2 Códigos de Estado

B.2.1 Informativo

100 Continuar
101 Mudando Protocolos

B.2.2 Sucesso

200 OK
201 Criado
202 Aceite
203 Não autorizado
204 Nenhum conteúdo
205 *Reset*
206 Conteúdo parcial

B.2.3 Redirecionamento

300 Múltipla escolha
301 Movido
302 Encontrado
303 Ver outro
304 Não modificado
305 Usar *proxy*
306 Não usado
307 Redirecionamento temporário

B.2.4 Erro de cliente

400 Pedido inválido
401 Não autorizado
402 Pagamento necessário
403 Proibido
404 Não encontrado
405 Método não permitido
406 Não aceitável
407 Autenticação de *proxy* necessária
408 *Timeout* do pedido
409 Conflito
410 Não disponível
411 Comprimento necessário
412 Pré-condição falhou
413 Entidade de solicitação muito grande
414 URI do pedido muito longo
415 Tipo de mídia não suportado
416 Intervalo solicitado não satisfatório
417 Falha na execução

B.2.5 Erro de Servidor

500 Erro interno no servidor
501 Não implementado
502 *Gateway* inválida
503 Serviço não disponível
504 Tempo limite de *gateway*
505 Versão HTTP não suportada

Anexo C

Comparação de *frameworks* REST

Tabela C.1: Tabela comparativa de *frameworks* REST - Parte 1

	Linguagem	**Baseada em Recursos**	**Serialização**	**Autenticação**	**Cache**
Django Rest Framework	Python	Sim	XML, JSON, YAML, HTML	Básica, Token, Sessão, OAuth, OAuth2.0 e outros	Sim
Flask-RESTful	Python	Sim	JSON	Não	Não
Restlet	Java	Sim	JSON, XML, CSV, YAML	Básica, Digest, Amazon S3, OAuth2.0	Não
Spark	Java	Não	Não	Não	Não
Sinatra	Ruby	Não	Não	Através de middleware Rack	Sim
Express	JavaScript	Não	JSON	Não	Não
Restify	JavaScript	Não	JSON, texto, octet-stream	Básica, através de assinatura	Sim
Sails	JavaScript	Sim	JSON	Através de extensões	Não
Loopback	JavaScript	Sim	JSON	Facebook, Google, Twitter, GitHub, OAuth2.0	Não

Tabela C.2: Tabela comparativa de *frameworks* REST - Parte 2

	Permissões	**Filtros**	**HATEOAS**	**Logging**	**Documentação**
Django Rest Framework	Sim	Sim	Sim	Não	Simples, intuitiva, detalhada com exemplos
Flask-RESTful	Não	Não	Não	Sim	Detalhada, com explicação através de exmplos
Restlet	Sim	Sim	Não	Sim	Bastante extensa e detalhada
Spark	Não	Sim	Não	Não	Simples e miimalista
Sinatra	Não	Sim	Não	Sim	Extensa e simples, com recursos a exemplos
Express	Não	Sim	Não	Sim	Detalhada e intuitiva, com exemplos práticos.
Restify	Sim	Sim	Não	Sim	Boa explicação de todas as funcionalidades disponíveis e alguns exemplos de utilização
Sails	Não	Sim	Não	Sim	Muito minimalista
Loopback	Sim	Sim	Não	Sim	Bastante detalhada

Anexo D

Relatório de Cobertura de Código

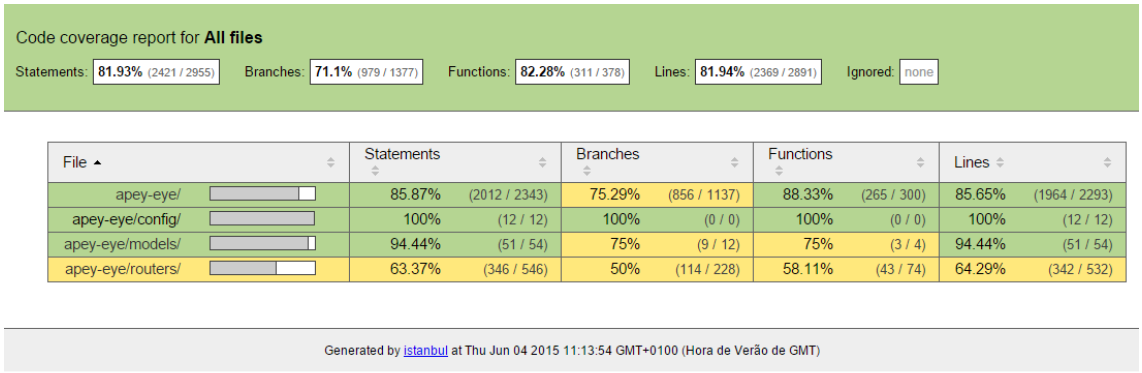

Figura D.1: Relatório de cobertura de código

Anexo E

Inicialização de serviços REST

E.1 Koa

```
1  var koa = require('koa'),
2      route = require('koa-route'),
3      app = koa();
4
5  app.use(route.get('/api/items', function*() {
6      this.body = 'Get';
7  }));
8  app.use(route.get('/api/items/:id', function*(id) {
9      this.body = 'Get id: ' + id;
10 }));
11 app.use(route.post('/api/items', function*() {
12     this.body = 'Post';
13 }));
14 app.use(route.put('/api/items/:id', function*(id) {
15     this.body = 'Put id: ' + id;
16 }));
17 app.use(route.delete('/api/items/:id', function*(id) {
18     this.body = 'Delete id: ' + id;
19 }));
20
21 var server = app.listen(3000, function() {
22   console.log('Koa is listening to http://localhost:3000');
23 });
```

Listing E.1: Inicialização de um serviço REST através de Koa

E.2 Hapi

```
1  var Hapi = require('hapi');
2  var server = new Hapi.Server(3000);
3
4  server.route([
5    {
6      method: 'GET',
7      path: '/api/items',
8      handler: function(request, reply) {
9        reply('Get item id');
10     }
11   },
12   {
13     method: 'GET',
14     path: '/api/items/{id}',
15     handler: function(request, reply) {
16       reply('Get item id: ' + request.params.id);
17     }
18   },
19   {
20     method: 'POST',
21     path: '/api/items',
22     handler: function(request, reply) {
23       reply('Post item');
24     }
25   },
26   {
27     method: 'PUT',
28     path: '/api/items/{id}',
29     handler: function(request, reply) {
30       reply('Put item id: ' + request.params.id);
31     }
32   },
33   {
34     method: 'DELETE',
35     path: '/api/items/{id}',
36     handler: function(request, reply) {
37       reply('Delete item id: ' + request.params.id);
38     }
39   }
40 ]);
41
42 server.start(function() {
43   console.log('Hapi is listening to http://localhost:3000');
44 });
```

Listing E.2: Inicialização de um serviço REST através de Hapi

E.3 Solução implementada

```
1  import ApeyEye from '../apey-eye';
2
3  let HapiRouter = ApeyEye.HapiRouter,
4      HapiGenericRouter = ApeyEye.HapiGenericRouter;
5
6  class ExampleResource extends ApeyEye.Resource{
7      constructor(){
8          super(async()=>{
9              return "POST"
10         })
11     }
12     static async fetchOne(options){
13         return "GET id:" + options.id;
14     }
15     static async fetch(options){
16         return "GET";
17     }
18     async put(options){
19         return "PUT id:" + options.id;
20     }
21     async delete(options){
22         return "DELETE id:" + options.id;
23     }
24 }
25
26 let router = new HapiGenericRouter();
27 router.register([{
28         path: 'restaurant',
29         resource: RestaurantResource
30     }]);
31
32 router.start({port: 3000}, function (err, server) {
33     if (!err) {
34         console.log('Server running at', server.info.uri);
35     }
36     else {
37         console.log('Error starting server');
38     }
39 });
```

Listing E.3: Inicialização de um serviço REST através da solução implementada